AD-A091 384 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
ON THE DESIGN, USE, AND INTEGRATION OF DATA MODELS.(U)
MAY 80 R A EL-MASRI MDA903-76-C-0206
UNCLASSIFIED STAN-CS-80-801 NL

1 OF 2
AD A
091384

12

Stanford Department of Computer Science
Report No. STAN-CS-80-801 14

11 May 1980

12 120

6

ON THE DESIGN, USE, AND INTEGRATION OF DATA MODELS.

by

10 Ramez Aziz El-Masri

9 Doctoral thesis,

Research sponsored by

Advanced Research Projects Agency

15 DA903-76-C-0206

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
STAN-CS-80-801

2. GOVT ACCESSION NO.
AD-A091384

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
On the Design, Use, and Integration of Data Models

5. TYPE OF REPORT & PERIOD COVERED
technical, May 1980

6. PERFORMING ORG. REPORT NUMBER
STAN-CS-80-801

7. AUTHOR(s)
Ramez Aziz El-Masri

8. CONTRACT OR GRANT NUMBER(s)
MDA903-76-C-0206

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Department of Computer Science
Stanford University
Stanford, California 94305 USA

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Avenue, Arlington, Virginia 22209

12. REPORT DATE
May 1980

13. NO. OF PAGES
220

14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office)
Mr. Philip Surra, Resident Representative
Office of Naval Research, Durand 165
Stanford University

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)
Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In spite of the large number of data models that have been proposed, no consensus exists about what concepts are really important for data modelling. Presentations of the various models cite the advantages of their approach, and discuss what the model best can represent. The advantages cited vary widely, and are usually very difficult to measure. How can we measure advantages of different approaches when they include ease of representation, ease of implementation, naturalness of expression, power of query language, non-procedurality of query language, expression of structural constraints, efficiency of implementation, formal foundations, support of existing implemented models, ... etc?

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Another problem which we encounter when we attempt to compare models is that not all models are defined at the same level. For example, hierarchical and network models include numerous implementation details. The relation model does not include any inter-relation constraints. Although some real-world constraints are captured by functional, multi-valued, and other dependencies, no systems represent these constraints except the functional dependencies that result form the primary keys of relations. The entity-relationship model is claimed to be a real-world model, rather than an information structure model, as are semantic and binary relationship models – does that imply these models cannot or should not be used for database implementation?

In this thesis, we are concerned with data models at the information structure level, rather than at the real-world semantic level. However, we want the information structure data model that we use to be able to capture many of the semantic concepts of the real-world that it models, as well as to support a query language which is close to the way users perceive the real-world to be organized (to a certain extent). Hence, our first task is to define the real-world semantics that we want to correctly represent in our model.

In Chapter 2 of the thesis, we define the real-world concepts we want to model. The concepts in this chapter are mostly from the literature of the semantic database models, and include the majority of concepts discussed therein. For each concept discussed, we will specify the origin (as far as we know) of that concept. The characterization of structural properties of relationships in Section 2.2 though is partly our own. The real-world concepts presented in Chapter 2 can also be used as a basis for comparing the representational powers of various models, although we do not attempt to compare models in this thesis.

Next, in Chapter 3, we discuss the information structure model that we use in this thesis, the *Structural Model*. The structural model is based upon the relational model. The main extension to the relational model is the concept of *logical* or *structural connections* between relations. The connections are used to represent structural properties in relationships as well as subclass concepts, which we will discuss in Chapter 2. The ideas behind this model originated from the model in Wiederhold, Chapter 7. However, the formal definitions are our own.

Connections are also used to define a simple, object-oriented language for query and update specification which is independent from the model structure to some extent. The language structure is quite simple, and we believe consistent with our desire for a high-level language that matches user expectations of the real-world (although we cannot prove this). We present the Structural Model Query Language (SMQL) in Chapter 4, as well as the language used for defining a structural model for a user, the Structural Model Definition Language (SMDL).

In Chapter 5, we show how the structural model can correctly represent the real-world concepts discussed in Chapter 2, using the small set of concepts of the model (domains, attributes, relations, and connections). The simplicity of the relational model is maintained to a certain extent, since the only additional concept introduced is that of connections. As we shall see, the value of these connections is twofold. First, they are used to represent the structural properties of relationships in the model. Second, they allow us to define the query language by giving names to the connections in both directions. A third advantage, no concered with modelling, but just as important, is that system implementors are aware of these logical connections, and can refer to them during file and access path implementation. The implementors are also made aware of the structural constraints specified by the connections, so they can consider efficient methods of checking these constraints. This is not possible if these constraints are defined in a general integrity assertion subysystem, separately from the model as in a generalized relational system.

The structural model also allows us to integrate different user data models to form an integraged database model which correctly supports the different user models. This allows the three-level database management system architecture discussed in Section 1.3. Data model integration, or view integration, has recently been discussed as a possible improvement on the traditional method for logical database design. However, integration has not been considered in detail. This thesis is the first attempt to examine the way in which data models can differ, and to discover methods to integrate the data models of different users. We will discuss data model integration in Chapter 6.

Finally, we give our conclusions and directions for future work in Ch

Stanford Department of Computer Science
Report No. STAN-CS-80-801

May 1980

ON THE DESIGN, USE, AND INTEGRATION OF DATA MODELS

by

Ramez Aziz El-Masri

ABSTRACT

In spite of the large number of data models that have been proposed, no consensus exists about what concepts are really important for data modelling. Presentations of the various models cite the advantages of their approach, and discuss what the model best can represent. The advantages cited vary widely, and are usually very difficult to measure. How can we measure advantages of different approaches when they include ease of representation, ease of implementation, naturalness of expression, power of query language, non-procedurality of query language, expression of structural constraints, efficiency of implementation, formal foundations, support of existing implemented models, ... etc?

Another problem ~~which we~~ encountered when we attempt to compare models is that not all models are defined at the same level. For example, hierarchical and network models include numerous implementation details. The relation model does not include any inter-relation constraints. Although some real-world constraints are captured by functional, multi-valued, and other dependencies, no systems represent these constraints except the functional dependencies that result form the primary keys of relations. The entity-relationship model is claimed to be a real-world model, rather than an information structure model, as are semantic and binary relationship models — does that imply these models cannot or should not be used for database implementation?

In this thesis, ~~we are~~ is concerned with data models at the information structure level, rather than at the real-world semantic level. However, we want the information structure data model that we use to be able to capture many of the semantic concepts of the real-world that it models, as well as to support a query language which is close to the way users perceive the real-world to be organized (to a certain extent). Hence, our first task is to define the real-world semantics that we want to correctly represent in our model.

In Chapter 2 of the thesis, we define the real-world concepts we want to model. The concepts in this chapter are mostly from the literature of the semantic database models, and include the

majority of concepts discussed therein. For each concept discussed, we will specify the origin (as far as we know) of that concept. The characterization of structural properties of relationships in Section 2.2 though is partly our own. The real-world concepts presented in Chapter 2 can also be used as a basis for comparing the representational powers of various models, although we do not attempt to compare models in this thesis.

Next, in Chapter 3, we discuss the information structure model that we use in this thesis, the *Structural Model*. The structural model is based upon the relational model. The main extension to the relational model is the concept of *logical* or *structural connections* between relations. The connections are used to represent structural properties in relationships as well as subclass concepts, which we will discuss in Chapter 2. The ideas behind this model originated from the model in Wiederhold, Chapter 7. However, the formal definitions are our own.

Connections are also used to define a simple, object-oriented language for query and update specification which is independent from the model structure to some extent. The language structure is quite simple, and we believe consistent with our desire for a high-level language that matches user expectations of the real-world (although we cannot prove this). We present the Structural Model Query Language (SMQL) in Chapter 4, as well as the language used for defining a structural model for a user, the Structural Model Definition Language (SMDL).

In Chapter 5, we show how the structural model can correctly represent the real-world concepts discussed in Chapter 2, using the small set of concepts of the model (domains, attributes, relations, and connections). The simplicity of the relational model is maintained to a certain extent, since the only additional concept introduced is that of connections. As we shall see, the value of these connections is twofold. First, they are used to represent the structural properties of relationships in the model. Second, they allow us to define the query language by giving names to the connections in both directions. A third advantage, no concered with modelling, but just as important, is that system implementors are aware of these logical connections, and can refer to them during file and access path implementation. The implementors are also made aware of the structural constraints specified by the connections, so they can consider efficient methods of checking these constraints. This is not possible if these constraints are defined in a general integrity assertion subsystem, separately from the model as in a generalized relational system.

The structural model also allows us to integrate different user data models to form an integraged database model which correctly supports the different user models. This allows the three-level database management system architecture discussed in Section 1.3. Data model integration, or view integration, has recently been discussed as a possible improvement on the traditional method for logical database design. However, integration has not been considered in detail. This thesis is the first attempt to examine the way in which data models can differ, and to discover methods to integrate the data models of different users. We will discuss data model integration in Chapter 6.

Finally, we give our conclusions and directions for future work in Chapter 7.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the Advanced Research Projects Agency of the Department of Defense under ARPA Order No. 2494, Contract MDA903-76-C-0206. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.

ON THE DESIGN, USE, AND INTEGRATION OF DATA MODELS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

by
Ramez Aziz El-Masri
May 1980

TO
Amalia and Ramy
AND
My Parents

ACKNOWLEDGEMENTS

I am very grateful to my thesis advisor, Professor Gio Wiederhold, for his guidance and help. This thesis would not have been written but for him. He introduced me to the area of databases, and guided me through the difficult stages of learning about a new subject. Many of the ideas presented in this thesis originated from discussions with him. He also helped me improve my writing skills.

I also wish to thank my thesis committee members, who suggested valuable improvements to the thesis. In particular, Professor Susan Owicki helped me with Chapter 4, and Dr. Daniel Sagalowics contributed tc the ideas presented in Chapter 5. Professor Jeffrey Ullman helped me learn more about database theory.

I am grateful to Professor Don Knuth for his encouragement and guidance, particularly during my first two years at Stanford, and to Professor Robert W. Floyd for supporting me for a year on the CHIRON project, through which I gained valuable experience.

My colleagues on the KBMS project provided valuable discussions and technical assistance, in particular Rich Bagley, Jim Davidson, Sheldon Finkelstein, Hector Garcia-Molina, Jerry Kaplan, Jonathan King, and Tom Rogers.

In addition, numerous people from the Computer Science Department have helped make my stay at Stanford enjoyable. These include Vacek Chvatal, Pat Jacobs, Thomas Lengauer, Jayne Pickering, Lyle Ramshaw, Steve Simon, Luis Trabb-Pardo, and Frank Yu.

I also wish to acknowledge some of the friends we made at Stanford. Foremost among them are Voy and Gio Wiederhold, and their family, who have helped me and Amalia on occasions too numerous to recount. Others include Safaa and Morad Aboseif, Ahmed Abuel-Haija, Hana and Omar El-Sawy, Teresa and Ray Faith, Hassan Khalil, Nadia and Said Khalil, and Ellen and Jonathan King.

The TT (Tuesday and Thursday) soccer gang provided me with an important outlet during some difficult times. I immensely enjoyed playing soccer, scoring goals, and arguing with them during my years at Stanford.

My parents, Asis and Dora, and my brother Maher, gave me continuous encouragement, love, and support during my stay at Stanford.

Finally, but most important, Amalia and Ramy provided me with love, inspiration, and happiness.

TABLE OF CONTENTS

1 INTRODUCTION

This thesis is concerned with the design and use of large, integrated, multi-user database systems. In this chapter, we define the type of database system we are concerned with, and discuss previous related work. We then specify the aim of this thesis.

1.1 MULTI-USER DATABASE SYSTEMS

A *generalized database system* is used to manage information about some aspect of the real world. To warrant the term generalized, the system should be independent of any particular application area, and the concepts and capabilities supplied by the system should be applicable to widely varying situations.

Database technology is geared to handle *structured* or *formatted* data, in the sense that large amounts of data are handled that have identical or similar structure. Hence, it is not applicable to managing information about situations where very little uniformity exists.

The predecessors of database systems are single-user file systems. A user who needs to store and access relatively large amounts of structured data uses the file system supplied by the computer installation. File systems provide access methods, so that structured data can be accessed more efficiently. System routines are available to do frequently required operations, such as sorting. However, access to the data to do retrieval or update always has to be accomplished by programs. These programs are written according to the needs of the user.

When several users need to use the same information, they can be granted access to the file at different times. This method of sharing data assumes that all the users agree on the information they need. If this is not possible, all the information the users needed must be stored even though some users require less data than others. This created problems of data *privacy* and *security*, where some of the data in the file is private to some users, but strongly related to the data of the other users and hence included in the same file. The low-level access to the file by programs made it difficult to enforce privacy and security for subsets of the data.

Another widely used method is to have each user have his own data on his own file, even though the different files include some data that is essentially duplicated. This led to *integrity* and *redundancy* problems. The integrity problem is that data which is common to several files represents the same information, so when updates to such data occur they have to be applied to all the files that include the data. Since users maintain their own files, this is not always possible. The redundancy problem is that the same

data is replicated, which requires several times the amount of computer storage than is really necessary.

The problems, discussed above, with file systems led to the development of generalized *database management systems*. A short history of the evolution of database management systems is given in [Sib76] and [FrSi76].

Another, perhaps more important, reason for the development of database systems is to provide a *high-level interface* to users. This interface hides from the users the details of how the information is stored, and provides the users with operations that match their perception of the data, rather than machine-oriented operations that reflect the way the data is stored. This led to the development of various techniques for *data modelling*.

Hence, our definition of *multi-user integrated database systems* is that they are computer software systems which are designed to facilitate the management and sharing of large amounts of structured data, and to provide general, high-level capabilities for data access to the users.

These are the types of database systems we are concerned with in this thesis, although many of the principles hold for small and medium size database systems, especially those principles concerned with the high-level user interfaces. These systems are used to manage large amounts of information, such as information of interest to large organizations, where many diverse users within the organisation are expected to use the database. The situation which this type of system is used to store and manage information about usually has the following properties:

(1) A large amount of information exists, which makes data management difficult and expensive without automation.

(2) The information has some regularity about it, which facilitates the grouping together of information of similar structure.

(3) Many groups of users are expected to use the database. Each group of users is concerned about the part of the total situation that is of interest to members of the group.

The first step in the design of a database is to examine, for each expected group of users, the part of the real-world situation of interest, and any available data that describes it, for regularities. Some simplifications are made to make the situation uniform and minimize exceptions. A *data model* is then designed to represent the data requirements of this group of users. Then, the numerous user data models are analysed to identify parts which are common to some of them, and are *integrated* into a global *database model* to support all the groups of users.

The modelling technique used is central to this process. We can only represent as much of the characteristics of a real-world situation as the formalisms of the data modelling technique allow us. A major part of this thesis is an attempt to discover a small set of primitives that are capable of representing the situations that occur frequently in database modelling. This is different from trying to represent *all* possible real-world semantics. What we attempt to accomplish is to identify a small number of concepts that correctly model the majority of situations relevant to database management.

Chapter 2 is an attempt to classify the situations of interest in database modelling. Clearly, this is not a well-defined topic, and arguments can be made to include many more concepts. However, our discussion includes most of the concepts in existing and proposed data models.

A brief note on terminology. The term *data model* is often used to mean two different concepts. The first concept refers to a model of the data that a group of users of the database are interested in. The second concept refers to a general approach to modelling data (the relational model, the entity-relationship model, the network model, the structural model, ... etc). Our use of the term will be clear from the context.

The next section is a brief survey of current modelling approaches.

1.2 A SHORT SURVEY OF MODELLING APPROACHES

Modelling approaches, or *database models* are used as a formal means of describing situations about which data is to be managed, and as a basis for implementing the generalised database management system. Hence, a database model should be able to describe correctly the concepts that appear frequently in the situations that are represented. The model should also be convenient to use for retrieval and update of the database by the users, and should make implementing the processes that handle the update and retrieval on the computer system feasible.

Hence, the important criteria for the choice of a formal database model are:

(1) Correctness: The model must be able to describe many of the semantic concepts needed for data modelling correctly. We will discuss these semantic concepts in Chapter 2.

(2) Simplicity: The model should not be so complex that it becomes difficult and awkward to use, both for data modelling and for database system implementation.

(3) Formality: The model should be based on some formal mathematical principles, since this allows processes that use the model to be formally specified and easier to verify.

(4) Implementation feasibility: An efficient implementation of a database system based on the model should be feasible.

(5) Data independence: Though point (4) is very important, it is useful to separate the data model from the implementation of the actual database storage and access structures, since both modelling and implementation are quite complex. This also allows restructuring of implemented access paths when the usage patterns of the database change without affecting the data model.

We will expand on these criteria in Section 3.1, after our discussion of the semantic concepts that we need to model real-world situations in Chapter 2.

The structured nature of the situations described by databases has, in the majority of data models, led to some separation between the description of the data and the actual data. Whenever some *instances* of data possess sufficient structural similarity, those instances are classified together into a *class* of data. The rules that the many instances of the class are expected to obey are specified once, in a *schema*.

Hence, the *schema* contains a description of the data. Each similarly structured class of data is represented in the schema by some *class descriptor* which specifies the structural and, in some cases, behavioural rules that each *instance* of the class is expected to follow. The schema also includes information about *relationships* between the classes of data. For example, it is possible that behavioural rules in the schema specify how instances from different data classes can be related.

Each instance of a class of data is formed from *data items*, which are values of data. The structure of the data class is usually specified by the *types* of data items each instance of the class is formed from. A *data item descriptor* is included in the class descriptor for each such data item. The data item descriptor usually includes a *name* and a *type* or *domain*, plus additional information that varies from one model to the next.

The terminology for class descriptors varies in different data models. In early data models (hierarchical, network) they are called *record types*, a term which reflects the dependence of these models on computer storage structures. In fact, these models were based on implemented systems, because implementations of hierarchical and network systems preceded the definition of these models.

In relational models, class descriptors are called *relation schemas* or *relation schemes*, which reflects the mathematical nature of the relational model. Other terminology includes *entity classes*, *object classes*, and *relationship classes* (in data models which differentiate between types of data classes), which reflects an attempt by these models to capture "semantics" of the situations being modelled.

Data item descriptors are called *data items* in hierarchical and network models, *attributes* in relation-based models, and *properties* in some other semantic models.

We now give a quick survey of data models from the literature. Readers who are not familiar with database models should skim through, or postpone reading this survey, and come back to it after they have read Chapter 2, since it may not make much sense to them now.

Many data models have been proposed. The earliest database systems were based on hierarchical storage structures, which led to the definition of the *hierarchical* data model. In this model, data classes are described by *record types*, which are organised hierarchically into tree structures. The *root* record type is the top node of the tree, and this root node owns one or more *dependent* record types, one level lower in the hierarchy, and so on. A record type description includes *data items*, that specify the type of data each record instance of the record type will include. The record instances are organised into hierarchical trees, with the root record as *owner* of the tree, and dependent records at lower levels of the tree.

The main relationships between record types are hierarchical in nature, and records at lower levels are owned by the records higher up the tree. Non-hierarchical relationships between record types have to be represented by *references* (or *pointers*) from other hierarchical tree structures in a non-symmetric fashion. Most hierarchical models have no formal means to represent references.

The best known implemented system using hierarchical concepts is IBM's IMS (Information Management System) [IBM75]. System-2000 [MRI72] of MRI is another example of a system that uses hierarchical concepts.

The *network* model allows representation of non-hierarchical relationships among record types. It is also based upon an implemented system, IDS [Hon72], which relied heavily on ring storage organisations. These organisations are more general, but also more complex than simple hierarchies. Bachman formalised the concepts used in this

system by proposing the data structure diagram technique for describing these ring structures [Bach69].

In the network model, the record types are organised as a network of record types which are connected by *set types* or *links* between record types. Each set type describes a 1:N connection between two record types: the *owner* record type and the *member* record type. A record instance from the owner record type is explicitly connected to sero or more record instances of the member record type.

Many network systems have been implemented based on the CODASYL (Committee on Data Systems Languages) DBTG (Database Task Goup) proposals [CODASYL71, CODASYL78], although most systems do not implement all of the features described therein. The CODASYL model and language originated from proposals to introduce database management facilities into the COBOL programming language, and are hence influenced by the structure of COBOL.

The retrieval and update languages for hierarchical and network models, called DML's [CODASYL74] (Data Manipulation Language), are procedural in nature, and hence programs must be written to carry out retrieval and update functions. Recently, attempts have been made to define non-procedural retrieval languages for these models [PaDaYu74, Brad78].

An introduction to the hierarchical and network models is given in [TaFr76] and [TsLo76], respectively. The CODASYL model is described in [CODASYL71] and a revised version appears in [CODASYL78].

The main disadvantages of database systems that have been implemented based upon the hierarchical and network models is that no clean separation exists between the model and the implementation in these systems. This is undoubtedly so because implementations preceded the model definitions. Also, since these were the earliest systems implemented, more recent concepts of subclasses are not supported by these models. Another criticism is that the procedural DML's require professional programmers to carry out even simple casual requests for data retrieval.

The *relational* model [Codd70] was introduced by Codd as a means of separating the data model from the implementation. The relational model is based on the mathematical theory of n-ary relations. Non-procedural, formal languages were proposed by Codd for data retrieval (the relational algebra [Codd72a] and the relational calculus language ALPHA [Codd71]).

In the relational model, an n-ary *relation* is used to represent a class of data. An n-ary relation is defined by n *attributes*. Each attribute is associated with a named *domain*, which specifies a set of data item values. The n-ary relation is a set of n-*tuples*. Each tuple in the relation is an instance of the class of data, and is composed of n values, one from each of the domains of the n attributes that define the relation. Hence, relations can be thought of as tables, where tuples are rows, and columns are attributes.

A data class of similar structure is described by a *relation schema*, and the instances of the class are the tuples within the relation. The attributes and domains describe the data items. Relationships among data classes are only implicitly represented, and can be discovered at retrieval time by matching attributes from different relations that have comparable domains. This matching is specified by the user.

The relational model has been subjected to intensive theoretical scrutiny. The concept of a *functional dependency* between sets of attributes was introduced by Codd to represent domain dependencies, and normal forms for relations were introduced to aid in the design of relational data models with favorable update properties [Codd72b, Codd74]. Multivalued dependencies where introduced by Zaniolo and Fagin [Zan76, Fag77] and a new fourth normal form which does not allow some third normal form relations with unfavorable update properties was defined in [Fag77].

Algorithms to synthesize third normal form relations from functional dependencies have been presented by Bernstein [Bern76] and Wang and Wedekind [WaWe75], although some of the problems related with the design were found to be NP-hard [BeBe79].

The relational model has been used as a basis of several database management systems (SYSTEM-R [Asea76], INGRES [HeStWo75], RISS [McMe75]). A survey of implemented relational systems appears in [Kim79]. Much research has been expended in the design of relational query languages for these systems. Relational languages include SQUARE [Boea75], QUEL [WoYo76], SEQUEL [Chea76], DEDUCE [Cha76], ALPHA [Codd71]. A survey and classification of relational languages appears in [Ull79], Chapter 4.

The main criticisms of the original relational model is that the uniform relational structures cannot represent sufficient information about the semantics of the real-world situation [SchSw75, CaKa76, McL78, Wied78]. In particular, semantic connections between relations are not represented, so the user must explicitly connect (or *JOIN*) tuples from related relation together when specifying a query. This has led to several systems which store this connection information separately from the model and use it in query processing (APPLE [CaKa76], IDA [Sag77]). These systems use a more natural query language without requiring the user to connect relations explicitly.

A related criticism holds for inter-relation update constraints. These cannot be represented in the relational model. Extensions to the relational model allow arbitrary integrity assertions to be specified on a relational model in an integrity subsystem, separate from the model itself [Ston74, McL76]. These assertions can be used to represent inter-relation update constraints, among other constraints. However, these are not part of the model itself and do not influence the data model design process. Maintaining arbitrary constraints in an implementation may become expensive as the number of constraints increases, and hence it is useful to include basic constraints in the data model so that the implementation designer is aware of them.

The original relational model does not support sub-class concepts. Smith and Smith [SmSm77] provided extensions to the model that take sub-class concepts into account. For other shortcomings of the relational model, see [McL78], Chapter 2.

The majority of implemented database systems are based upon one of the above models—hierarchical, network, and relational. However, many other data models have been proposed. Extensions to the relational model which allow representation of inter-relation constraints are given in [SchSw75, Chen76, Wied77 (Chapter 7)]. In particular, Chen's *entity-relationship* model [Chen76] has gained recognition because it can support both relational and network views of data to some extent.

In these models, relationship constraints are specified by the definition of types of relations, and recognising the inter-dependencies between relation types. These inter-dependencies can represent the inter-relation constraints from the real-world situation. A classification of inter-relation constraints, and a comparison of how they are represented in several data models appears in [ElWi80].

Another class of data models is based on binary relationships. These *binary relationship* or *binary association* models treat domains (classes of single data items) as their basic modelling primitives. No grouping together of strongly related data items into record types or relations is represented in these models. Rather, all relationships among these primitive domains are explicitly specified as binary relationships. These models include [Abr74, Falk76, BiNeu78, BrPaPe76].

Another class of database models are the *functional* data models, which are quite similar to binary relationship models [SiKe77, Ship79, BuFr79, HoWaYa79]. These models are binary relationship models, where the binary relationships are defined as single and multi-valued functions. Functional data definition and manipulation languages such as DAPLEX [Ship79] and FQL [BuFr79] have been proposed. These languages are claimed to be more natural to use than other non-procedural query languages, such as the relational languages.

The definition of even a simple data model becomes quite complex in a functional model due to the necessity of explicitly defining every binary relationship. In functional models, a binary relationship is defined twice: once by a function, and a second time as the inverse of that function. Functions in these models differ from mathematical functions in that both single-valued and multi-valued functions are allowed.

Recently, criticisms have been levied against record and relation based data models as being too restrictive to represent commonly occuring situations in data modelling [Kent79]. It was hinted that binary relationship models might be more suitable. However, we argue against this notion. The restrictions of record and relation-based models stem from the restrictions in existing models, rather than being an inherent weakness of the record and relation-based approach. We show in Chapter 5 how the structural model, a relation-based model which we use in this thesis, can overcome many of Kent's objections.

The concept of a *subclass* has been used in Artificial Intelligence for some time. It was introduced to database modelling by Smith and Smith within the context of the relational model [SmSm77]. More recent data models, which attempt to represent more of the semantics of the real-world situation, are given in [HaMc78, NaSc78]. Some concepts from the semantic data model of Hammer and McLeod [HaMc78, McL78] are presented in Section 2.1. It is a rich semantic model, with many explicit semantic types of object classes. However, it is not clear if these concepts are complete in any way, nor why they were selected and other concepts ommitted. We will show in Chapter 5 that these concepts can be represented by a small set of basic primitives of the structural model, which is used in this thesis.

The concept of *role* is introduced in [BaDa77], and discussed further in [Ship79, WeWo80]. We will discuss roles in Section 2.3.

Another class of data models includes models which do not have a clear distinction between the data and the data description. These include DIAM [Sen75] and the semantic network model [RoMy75], which is based upon a well known representation technique in Artificial Intelligence. However, these models are not well suited to the type of large, structured databases we are dealing with, since they do not take advantage of the regularity of the data.

1.3 DATABASE SYSTEM ARCHITECTURE

The architecture we assume for a database system is quite similar to that proposed by the ANSI/X3/SPARC (Standards Planning and Requirements) committee of the American National Standards Association [Ste75]. The architecture proposed therein was prompted by attempts to standardize the CODASYL model at a time when it was felt not enough was known about database models and systems to warrant standardization.

This *architecture is for multi-user database systems. Three main levels of* data descriptions exist:

(1) The *user* level: At this level, a data model exists for each group of users of the database that are all interested in a particular subset of the database. An *external schema*, or *user view*, exists to describe each of these models. The user group views the database as though it contained only the data described in their external schema.

(2) The *global* level: The global level includes a complete description of the database in a *conceptual* or *global* schema. The conceptual schema hence includes sufficient information to support all external schemas, although external schemas do not have to be identical to a subset of the conceptual schema.

(3) The *implementation* level: A description of how the data described in the conceptual schema is stored on the computer system. This implementation description is in the *internal* schema.

Figure 1 shows this architecture.

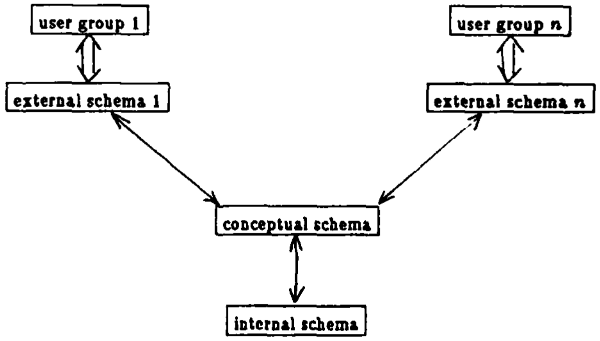


Figure 1 The ANSI/X3/SPARC database management system architecture

At both the external and conceptual levels, the information descriptions are oriented towards the users' perspective of the situation. Hence, the models used at these levels should be able to express the concepts that occur frequently in real-world situations described by databases.

A mapping exists between the external and conceptual levels to define the correspondence between the structures at each level. These mappings can be simple if an external schema is only a subset of the conceptual schema. When the external schema does not correspond directly to a subset of the conceptual schema, the mappings become more involved. Note that this latter case is allowed as long as an external schema does not include more information than exists in the conceptual schema.

The internal schema describes how the data is organised into files, and what auxiliary access structures are available to aid in efficient access to the data. The conceptual to internal mapping defines the correspondence between the actual stored data and the high-level descriptions of the conceptual schema.

This architecture supports *data independence* since, theoretically at least, there need be no dependency between the high-level data descriptions of the external and conceptual schemas, and the way the data is actually stored as described in the internal schema. This is important when many classes of data exist in the database, with intricate relationships between them. *Auxiliary access structures* can be implemented to support more efficiently the prevailing access paths. When usage patterns change, the access structures and, perhaps, the file organisations can be changed without having to change the high-level user descriptions.

This is particularly important since on the initial design of the database, only estimates of the usage patterns are available. If estimates are innaccurate, this can lead to an implementation where frequently used access paths are quite inefficient, which causes poor system performance. This is particularly true of very large databases.

The three-level architecture can also support *self-organizing* or *adaptive* database systems, where the system can keep usage statistics and automatically trigger reorganisation of access paths to match the currently prevalent access patterns. The implementation of this type of system is still in the research stage.

Finally, this type of architecture is particularly suitable for support of *distributed database systems*, where the data is stored in several nodes of a computer network. The user view need not be aware of the location where the actual data is stored. The mapping from the conceptual to internal levels can keep track of the distribution. Multiple internal schemas can be used.

The traditional view of database design is to first design the conceptual model of the data. This is usually accomplished by the *database administrator* of the organisation [DeBJo77], who is responsible for defining the data processing needs of the whole organisation. When user groups need access to the database, the database administrator can grant them access by defining an external schema.

Recently, a different approach has been suggested [NaSc78, ElWi79, WiEl79a, WiEl79b] which is aimed at better support of the different users' requirements. In

this approach, the expected user groups of the database first define their data models, with the aid of the database designers, to match their requirements and perception of the real-world situation that they are interested in. An *integration* phase follows, where common data represented in the user models is recognized, and a global database model is designed to support the user data models. The advantages of this *integration* approach are:

(1) Users have more freedom to define their data models than in the previous approach where they examine a database model and define their models based on it.

(2) The global database model is more explicitly defined to take into account the differences between the data models. This can simplify the mapping from external to conceptual levels.

(3) The global database model can support complete update capabilities for the user data models, which is not always possible in the other approach (see [DaBe78], [Asae76]).

(4) Errors in the way users perceive a given situation can be discovered when two user data models represent the same situation differently.

In Chapter 6, we discuss the ways in which data models can differ, and present a methodology for data model integration.

1.4 PURPOSE AND ORGANIZATION OF THESIS

In spite of the large number of data models that have been proposed, no consensus exists about what concepts are really important for data modelling. Presentations of the various models cite the advantages of their approach, and discuss what the model best can represent. The advantages cited vary widely, and are usually very difficult to measure. How can we measure advantages of different approaches when they include ease of representation, ease of implementation, naturalness of expression, power of query language, non-procedurality of query language, expression of structural constraints, efficiency of implementation, formal foundations, support of existing implemented models, ... etc.?

Another problem which we encounter when we attempt to compare models is that not all models are defined at the same level. For example, hierarchical and network models include numerous implementation details. The relational model does not include any inter-relation constraints. Although some real-world constraints are captured by functional, multi-valued, and other dependencies, no systems represent these constraints except the functional dependencies that result from the primary keys of relations. The entity-relationship model is claimed to be a real-world model, rather than an information structure model, as are semantic and binary relationship models—does that imply these models cannot or should not be used for database system implementation?

In this thesis, we are concerned with data models at the information structure level, rather than at the real-world semantic level. However, we want the information structure data model that we use to be able to capture many of the semantic concepts of the real-world that it models, as well as to support a query language which is close to the way users perceive the real-world to be organised (to a certain extent). Hence, our first task is to define the real-world semantics that we want to correctly represent in our model.

In Chapter 2 of the thesis, we define the real-world concepts we want to model. The concepts in this chapter are mostly from the literature of the semantic database models, and include the majority of concepts discussed therein. For each concept discussed, we will specify the origin (as far as we know) of that concept. The characterisation of structural properties of relationships in Section 2.2 though is partly our own. The real-world concepts presented in Chapter 2 can also be used as a basis for comparing the representational powers of various models, although we do not attempt to compare models in this thesis.

Next, in Chapter 3, we discuss the information structure model that we use in this thesis, the *Structural Model*. The structural model is based upon the relational model. The main extension to the relational model is the concept of *logical* or *structural connections* between relations. The connections are used to represent structural properties of relationships as well as subclass concepts, which we will discuss in Chapter 2. The ideas behind this model originated from the model in Wiederhold [Wied77], Chapter 7. However, the formal definitions are our own.

Connections are also used to define a simple, object-oriented language for query

and update specification which is independent from the model structure to some extent. The language structure is quite simple, and we believe consistent with our desire for a high-level language that matches user expectations of the real-world (although we cannot prove this). We present the Structural Model Query Language (SMQL) in Chapter 4, as well as the language used for defining a structural model for a user, the Structural Model Definition Language (SMDL).

In Chapter 5, we show how the structural model can correctly represent the real-world concepts discussed in Chapter 2, using the small set of concepts of the model (domains, attributes, relations, and connections). The simplicity of the relational model is maintained to a certain extent, since the only additional concept introduced is that of connections. As we shall see, the value of these connections is twofold. First, they are used to represent the structural properties of relationships in the model. Second, they allow us to define the query language by giving names to the connections in both directions. A third advantage, not concerned with modelling, but just as important, is that system implementors are aware of these logical connections, and can refer to them during file and access path implementation. The implementors are also made aware of the structural constraints specified by the connections, so they can consider efficient methods of checking these constraints. This is not possible if these constraints are defined in a general integrity assertion subsystem, separately from the model as in a generalised relational system [Ston74, McL76].

The structural model also allows us to integrate different user data models to form an integrated database model which correctly supports the different user models. This allows the three-level database management system architecture discussed in Section 1.3. Data model integration, or view integration, has recently been discussed as a possible improvement on the traditional method for logical database design [NaSc78, ElWi79]. However, integration has not been considered in detail. This thesis is the first attempt to examine the way in which data models can differ, and to discover methods to integrate the data models of different users. We will discuss data model integration in Chapter 6.

Finally, we give our conclusions and directions for future work in Chapter 7.

2 REAL-WORLD STRUCTURES AND RELATIONSHIPS

In this chapter, we discuss the concepts that are used to describe and simplify real-world situations that are often represented in database systems. The information to be represented is classified using these concepts, and the actual database is expected to follow the rules specified by these concepts.

The concepts described here are an attempt to categorise the important semantic concepts of database modelling that have been discussed in the literature of semantic data models. The discussion is informal, but in the final section of this chapter, we give a simple formal model to cover these concepts.

The concepts described here are those which a user would comprehend, and be comfortable with. Hence, it is useful for a database language to be based upon these concepts, since this would simplify the expression of queries for users of the database system. However, the ideas and concepts presented in this chapter are not formal enough to base an implementation upon them. In particular, the properties of object classes (see Section 2.1) are often repeated redundantly as properties of several object classes. As we shall see in Chapter 5, when a formal data model is designed we must to distinguish between basic and redundant data to maintain the data consistent. The formal data model should include the capabilities to express these concepts correctly to maintain the data consistent, and at the same time offer the capabilities for a high-level language based on the real-world concepts that make the interaction of the user with the database simple.

In Chapter 4, we present the Structural Model Query Language. The form of a query in this language is quite close to the concepts we discuss in this chapter. In Chapter 5, we show how the Structural Model, which we present formally in Chapter 3, can correctly represent the concepts presented here, and how the model can distinguish between basic and redundant representations.

Several concepts covered in this chapter have been discussed in the literature. The characteristics of *properties*, or *attributes*, of object classes that we discuss in Section 2.1 are in a sense "well known". In particular, the Semantic Data Model of Hammer and McLeod [HaMc78, McL78] includes a classification of the types of properties.

The concept of a *subclass*, discussed in Section 2.1.3, was first considered in the database environment by Smith and Smith [SmSm77] within the context of the relational database model. However, this concept has been used in the AI (Artificial Intelligence) field for a long time (the *ISA* hierarchy).

The classification of the *structural properties* of relationships, given in in Section 2.2, is our own. It is equivalent to a classification of binary relationships given by Abrial [Abr74] as we will show in Section 2.2.2.5. The concepts of relationships between roles rather than object classes, discussed in Section 2.3, is due to Bachman and Daya [BaDa77].

Throughout the discussion of real world concepts in this chapter, we will use examples from two situations to illustrate the concepts: a company situation, and a ships monitoring situation.

2.1 OBJECT CLASSES

The real-world situation consists of ***objects***, their ***properties***, and their ***relationships*** with other objects. We live in a structured world, so usually many objects exist that serve similar functions, and are identical in structure. Such groups of similar objects can be grouped into classes.

An ***object class*** is hence a class of objects of identical structure. By identical structure, we mean that each object in the class is described by a set of properties that are common to all objects in the class.

2.1.1 Properties of object classes

A ***property*** of an object class is used to describe some aspect of each object in the object class. Each object in the class possesses a *value* for each property of the class. This value describes the aspect of the object characterised by the property. A ***value set***, which contains all the values the property can take for particular objects in the class, is associated with each property.

Figure 2 shows some object classes and their properties for the company situation. Properties that can have many values for a single object are enclosed in braces ({ }). In our text and diagrams, we will use capital boldface to denote object classes, lower case boldface to denote properties, and italics to denote values.

Consider the EMPLOYEES class. An employee object has a value for each of the properties used to describe the class. A certain employee might be described by the values of properties shown in Figure 3.

Sometimes, a particular object does not have a value for one of the properties. For example, the employee *James King* does not have a supervisor, so the value *none* is entered for property supervisor. Other properties may be unknown for a particular object. Some of the properties are not allowed to have values *none* or ***unknown*** for any object in the class. We call these properties ***mandatory*** properties. For example, if every employee must have a name, number, and department, the properties **name**, **number** and **department** for the EMPLOYEES class are mandatory. Properties that are not mandatory are ***optional*** properties.

object class	properties
EMPLOYEES	**number, name, address, age, birth-date, sex, salary, {salary-history}, job, {job-history}, department, supervisor, {projects}**
DEPARTMENTS	**name, manager, location, {employees}, {projects}**
CHILDREN	**name, {parent-name}, age, birth-date, sex**
PROJECTS	**name, manager, {departments}, {employees}**

Figure 2 Some object classes and their properties from a company situation

number:	*556-33-9123*
name:	**(first:** *James*, **middle:** *Arthur*, **last:** *King*)
address:	**(street-name:** *Sanches Way*, **number:** *1154*, **apartment-number:** *none*, **town:** *Redwood City*, **state:** *California*, **sip-code:** *94061*)
age:	*29*
birth-date:	**(month:** *August*, **day:** *19*, **year:** *1950*)
sex:	*male*
salary:	*22312*
salary-history:	{(**start-date:** (**month:***September*, **year:** *1973*), **salary:** *17,500*), (**start-date:**(**month:** *January*, **year:** *1975*), **salary:** *19,999*), (**start-date:** (**month:** *December*, **year:** *1977*), **salary:** *22,312*)}
job:	*design engineer*
job-history:	{(**start-date:** (**month:** *September*, **year:** *1973*), **job:** *design engineer*)}
department:	*design*
supervisor:	*none*
projects:	{*X110, K55*}

Figure 3 Values of the properties for the employee James King

An object exists independently of the properties that describe it. However, in most cases, a *single property*, or sometimes a set of properties, exists whose values are unique for each object in the class. Such properties, or sets of properties, are called ***unique*** or ***identifying*** properties of the object class. For example, if every employee object must have a unique employee number, then the number property is an indentifying property for the EMPLOYEES class.

Sometimes, it is difficult to find a set of identifying properties for a class. For example, consider the CHILDREN class of Figure 2. The child name alone (we assume it is the proper name only here) is not unique among children. If we assume two children of an employee cannot have the same name, the set of properties {**name, parent-number**} may seem to be a set of *identifying* properties. However, this is true if only one of the parents works in the company.

Properties may be composed of simpler properties. For example, the **address** property of EMPLOYEES is composed of **street-name, number, apartment-number, town, state,** and **sip-code**. We call such properties ***compound*** properties, while non-decomposable properties are called ***simple*** properties. Compound properties can be composed of other compound properties. For example, in Figure 3, the **salary-history** property of EMPLOYEES is composed of the two properties **start-date** and **job**, and **start-date** is further composed of **month** and **year**.

A property may be single-valued or multivalued. A ***single-valued*** property has only one value of that property for each object in the class. A ***multi-valued*** property (also called a *repeating* property in the literature) may have a set of values of the property for a single object. Multi-valued properties of EMPLOYEES are **projects, salary-history,** and **job-history**, and are shown in braces in Figure 2.

We define a multi-valued property to be mandatory if it has at least one value of the property that is neither *none* nor *unknown* for each object in the class. The number of values a multi-valued property can take for a single object may be more precisely specified if more information is known about the situation. For example, if a company rule specifies that an employee must work on a project at all times, but cannot work on more than five projects at a time, the multi-valued property projects of EMPLOYEES may be further specified by two numbers: maximum 5, *minimum* 1 which limit the number of values that a single object can take for that property to between 1 and 5 values.

Some properties, in addition to providing values that describe an object, serve to relate this object with objects from other classes. We call these *relating* properties, and examples are the department and projects properties of *EMPLOYEES*. We will further discuss relationships in Section 2.2. Properties that are not relating will be called *describing* properties.

Finally, a property may have a value that is derivable from other properties of the object, or for that matter from properties of a related object or set of objects in another class. For example, the value of age for an employee is derivable from the value of birth-date. We call these properties *derivable* properties.

Table 1 shows the different characteristics of a property of an object class.

	characteristic	alternative	description of characteristic
(a)	unique	non-unique	every object in the class has a unique value for that property
(b)	mandatory	optional	every object must have a value that is neither none nor unknown for that property
(c)	simple	compound	the property is not decomposable into other properties
(d)	single-valued	multi-valued	only a single value of the property is allowed for each object in the class
(e)	*relating*	describing	the value of the property serves to relate an object to another object (in addition to describing the object)
(f)	derivable	non-derivable	the value of the object can be derived from values of other properties of the object

Table 1 Characterization of properties of an object class

2.1.2 Semantic types of object classes

It has been suggested that classes of objects can be differentiated into semantic types. Our discussion here follows that of McLeod [McL78]. However, these semantic types of classes have a similar structure, and hence are not differentiated in the structural model, which is the model we use in this thesis (see Chapter 3).

It is clear that event classes (see Section 2.1.2.4 below) have a specific dependence on time. This makes their use in queries subject to our assumptions about time orderings. However, this aspect applies to ordered domains in general, and not only to time.

The distinctions into abstraction, and aggregation classes (see below) are not important in our view, since they are only special cases of relationships (see Section 2.2, and the way abstraction and aggregation classes are represented in the structural model in Section 5.4.2).

2.1.2.1 *Entity classes*

Classes of objects that have a physical or concrete existence in the real world situation are called *entity* classes. These kind of classes are called concrete object classes in [McL78]. *All the classes shown in Figure 2 are entity classes.*

2.1.2.2 *Abstraction classes*

Concepts may exist in the real-world situation that group objects from an object class together according to a set of properties of the class that have identical values for each object in the group. We call these *abstraction* classes. An abstraction class is defined over another object class, or several object classes. The properties that have common values are the properties of the abstraction class, and a single object in the abstraction class exists for each group of objects with identical values for these properties.

object class	properties
SHIPS	ship-id, ship-name, hull-number, ship-class, owner, country-of-registry
SHIP-CLASSES	ship-class, length, draft, dead-weight, gross-weight, ship-type
SHIP-TYPES	ship-type, {cargo-types}
VOYAGES	ship-id, departure-port, departure-time, destination-port, arrival-time
ARRIVALS	ship-id, port-name, arrival-time, dock-number
PORTS	port-name, country, number-of-docks, maximum-draft, maximum-length, {ships-currently-at-port}
CONVOYS	convoy-name, {ships-in-convoy}

Figure 4 Some object classes from the ships monitoring situation

Consider the classes in Figure 4 that are part of a ships-monitoring situation. An abstraction class SHIP-CLASSES describes properties that are common to groups of ships built to the same specification. Hence, an object in SHIP-CLASSES describes the length, draft, dead-weight, gross-weight, and ship-type properties of a group of ship objects. This group will consist of only one ship if it is the only one built to these specifications.

Another abstraction class SHIP-TYPES groups together objects from the abstraction class SHIP-CLASSES according to the types of cargoes they can carry. Note that SHIP-TYPES is a second-level abstraction: it also groups ship objects according to the types of cargoes they can carry.

2.1.2.3 Aggregate classes

An *aggregate* class is a class of objects, each of which consists of a group of objects from another class. It is different from abstraction classes in that it does not specify any additional properties for the group of objects that can actually be considered descriptions of the objects. Rather, each object of an aggregate class is made up of a group of objects of the other class. An aggregate class is quite similar to an abstraction class in the way one object of the abstraction or aggregate class groups together a number of objects of the other class.

An example is the CONVOYS class of Figure 4. Each convoy object specifies a group of ships that are in that convoy. Identifying aggregate classes allows reference to a particular aggregate by name, as well as to the objects that constitute the aggregate.

2.1.2.4 Event classes

A real-world situation often includes events that happen, in which one or more objects participate. Occurrences of a particular type of event form an *event* class. It is important to identify events because time plays an important role in the ordering of events. Hence, an intelligent user interface (one which we are not concerned with in this thesis) should take into account the way user queries interact with time.

Examples of event classes from Figure 4 are VOYAGES and ARRIVALS. We can identify two types of events: *point* events occur at a specific time (for example, an arrival) while *duration* events occur over a period of time (for example, a voyage).

Explicit identification of event classes implies an ordering of events by time of occurrence. For duration events, two such orderings are implicit: one for the starting time and one for the ending time of particular events.

2.1.3 Subclasses

A *subclass* is a subset of objects from a *base* class that are semantically distinguishable from the other objects in the base class. Usually, objects in the subclass satisfy some conditions that other objects in the class do not satisfy. For example, the objects in the subclass may have additional properties that describe them that are not applicable to objects not in the subclass.

Subclasses are also important because users will often refer repeatedly to a subclass of objects they may be intereseted in. A user interface should provide for such references. When the focus of a user switches to another subclass, the interface should allow him easy reference to the new subclass of interest.

Many subclasses can exist for a single object class. Every property of a class implicitly defines a set of subclasses where each subclass includes all the objects that have the same value for that property. If the property is single-valued, the set of subclasses is mutually exclusive (disjoint). For example, the job property of EMPLOYEES (Figure 2) implicitly defines a set of job subclasses where each subclass contains the employees that have the same job. If the property is single valued and unique, each subclass in the set contains one object.

A multi-valued property defines a set of subclasses that are not disjoint. An example is the projects property of EMPLOYEES which defines subclasses of employees that work on the same project. This is because the multi-valued property describes the fact that an employee can work on multiple projects.

Abstractions, aggregations and relationships with other object classes also define subsets of objects from a class: those objects that are grouped together by belonging to the same abstraction or aggregate object, or by being related to the same object from the other object class. For example, each ship-class object (abstraction) defines the subclass of ships of this class, each convoy object (aggregate) defines the subclass of ships that are in that convoy, and each department object (relationship) defines the subclass of employees that work in that department.

All the subclasses discussed so far are implicitly defined, since we can always create the subclass from the existing properties of objects. Other subclasses have to be explicitly defined because the objects in the subclass have additional properties that are not applicable to other objects in the class. These explicit subclasses express a rule that only objects in the subclass posses certain properties. Such rules are important to identify so that one can enforce them on the individual objects.

For example, the subclass MANAGERS of EMPLOYEES may have the additional properties managerial-rank and projects-managed. A further subclass of managers, DEPARTMENT-MANAGERS, may have an additional property department-managed as shown in Figure 5. Only employee objects that are managers, and hence belong to the MANAGERS subclass can have values for these additional properties.

subclass	base class	type	condition	additional properties
MANAGERS	EMPLOYEES	restriction	job=*manager*	managerial-rank, {projects-managed}
DEPARTMENT-MANAGERS	MANAGERS	non-restriction	none	department-managed

Figure 5 Subclasses of EMPLOYEES and MANAGERS

Note that in this example, both the projects-managed property of MANAGERS, and the department-managed property of DEPARTMENT-MANAGERS are relating properties.

Two types of subclasses can be defined. A *restriction subclass* is specified by restricting the values of a particular property (or set of properties) from the base class to a subset of the value set for that property. All objects in the base class that have values for that property that are in the specified subset of values are automatically included in the subclass.

A *non-restriction* subclass is not automatically specified by values of existing properties for objects in the base class. Rather, all objects in a non-restriction subclass have to be individually and explicitly specified. In the example of *Figure 5*, MANAGERS is a restriction subclass, since the property job already existed for the EMPLOYEES class. However, DEPARTMENT-MANAGERS is not a restriction subclass since no property existed to specify the restriction.

It can be argued that we may always introduce a property upon which the restriction can be specified. For example, a property department-manager-or-not can be introduced to the MANAGERS object class, which has the two values *yes* and *no*, depending upon whether a manager is a department manager or not. In our distinction, restriction subclasses are based on properties that already existed.

An object in the subclass inherits all the properties of the object from the base class. Hence, a manager object will also have all the properties of EMPLOYEES that are shown in *Figure 2*.

2.2 RELATIONSHIPS

We now discuss relationships between object classes, which leads us to another type of object class, the relationship object class, which we discuss in Section 2.2.1. In S-ction 2.2.2, we define and discuss the structural properties of a relationship, and in Section 2.2.3, we discuss relationships between categories of object classes, which leads us to the concept of the *role* of an object class.

A *relationship* between two object classes is a mapping that relates each object of one object class to a number of objects (possibly none) of the other object class. For example, a relationship DEPARTMENTS:EMPLOYEES relates each employee to *his department*. The same relationship also relates each department to its employees. Hence in Figure 2, the two relating attributes—department of the EMPLOYEES class and employees of the DEPARTMENTS class—describe the same relationship.

More than one relationship can exist between the same two object classes. For example, consider the two object classes SUPPLIERS that supply parts to a company, and PARTS that are used by the company. Two relationships may exist between these two object classes. One relationship relates each supplier with the parts he can supply, and the other relationship relates each supplier with the parts he is currently supplying Hence, it is useful to give a particular relationship a name. In this example, the first relationship can be called CAN-SUPPLY, and the second relationship can be called CURRENT-SUPPLY. Figure 6 shows this example.

A relationship has rules that govern the mapping of objects from the two object classes. We shall call these rules the *structural properties* of the relationship, and discuss them in Section 2.2.2. The relationship may also have describing properties. For example, the CURRENT-SUPPLY relationship may have a describing property number-of-parts for each (supplier, part) pair of objects that are mapped together by the relationship.

Relationships may also exist between more than two object classes. Consider a relationship between SUPPLIERS, PARTS, and PROJECTS that maps together a (supplier, part, project) triple when the supplier object in the triple currently supplies the specified part object to the specified project object. Now, it becomes more difficult to describe the relationship by relating properties of the object classes SUPPLIERS, PARTS and PROJECTS. However, it is straightforward to describe the relationship if we include a new type of object class, the relationship class.

object class	relating properties
SUPPLIERS	{parts(can be supplied)},{parts (currently supplied)}
PARTS	{suppliers(can supply)},{suppliers (currently supplying)}

Figure 6 Two relationships between SUPPLIERS and PARTS

2.2.1 Relationship classes

A *relationship class* between two object classes is a class of *relating* objects. Each relating object serves to relate two objects, one from each of these two classes, together. These two classes can be of any type, including other relationship classes. Since each relating object in the relationship class relates two objects together, it is identified by the two objects. Hence, some set of identifying properties from each of these two objects should be included in the properties of the relationship class.

A relationship among n object classes can always be decomposed into $n-1$ binary relationships, each of which is between only two object classes. This is easily verified. Choose any two of the object classes that participate in the relationship, and define a relationship class between them that includes all combinations which appear in the general relationship. Then, define another relationship class between this relationship class, and another of the object classes which includes all combinations that appear in the general relationship, and so on. Each step reduces the remaining object classes by 1, except the first step which reduces it by 2. Hence the $n-1$ binary relationships.

The number of binary relationships in a decomposition can be more than $n-1$ if the semantics of the situation cause natural decompositions of binary relationships between the object classes. An example is shown in Figure 7, in which all the relationships between SUPPLIERS, PARTS, and PROJECTS that we discussed earlier are shown.

The example shown in Figure 7 shows how the ternary relationship that describes the suppliers that currently supply parts to projects is decomposed. The relationship class CURRENT-SUPPLY describes the parts currently supplied by each supplier (which is a subclass of the CAN-SUPPLY relationship, since a supplier cannot be currently supplying a part which he has no capacity to supply). The relationship class PROJECTS-PARTS describes which parts a particular project uses. Finally, the relationship class PROJECT-SUPPLY describes the ternary relationship, which is now a binary relationship between the two relationship object classes CURRENT-SUPPLY and PROJECTS-PARTS. In this case the ternary relationship was decomposed into n binary relationships ($n = 3$) rather than $n-1$, which would have been the minimal decomposition.

object class	properties
SUPPLIERS	name, address
PARTS	name, number, description
PROJECTS	name, manager
CAN-SUPPLY	supplier-name, part-number
CURRENT-SUPPLY	supplier-name, part-number, quantity
PROJECTS-PARTS	project-name, part-number
PROJECT-SUPPLY	project-name, part-number, supplier-name

Figure 7 Relationships between SUPPLIERS, PARTS, and PROJECTS

The way a relationship is decomposed depends upon the semantics of the situation. *In this case, it was straightforward because the two binary relationships* CURRENT-SUPPLY and PROJECTS-PARTS from which the ternary relationship PROJECT-SUPPLY is composed also exist in our situation, and the semantics of the situation define the composition. In other cases, it may be necessary to define relationships which do not have a direct semantic correspondence in the real-world situation.

2.2.2 Structural properties of relationships

Usually rules exist, defined by the semantics of the real-world situation, that govern the behaviour of relationships. For example, consider the relationship DEPARTMENTS: EMPLOYEES that relates a department with the employees that work in it. In a particular company, the following two rules may hold for the relationship:

(1) Every employee must be related to exactly one department.

(2) A department is related to at least one employee, but can be related to any number of employees.

These two rules imply that the relationship is a *mutual* or *total dependency*, since the existence of an employee in the company situation depends upon his being related to a department, and the existence of a department depends upon its having at least one employee. The rules also imply that a department can be related to N employees, where N is unspecified. Hence, the *cardinality* of the relationship DEPARTMENTS:EMPLOYEES is 1:N.

We call such rules the *structural properties* of a relationship. Two structural properties can be identified: the cardinality and the dependency. We will only consider these properties for binary relationships, since relationships between n object classes are decomposable into $n-1$ or more binary relationships as discussed in the previous section.

The *cardinality* property of a relationship places restrictions on the number of objects of one class that may be related to an object of the other class. The *dependency* property specifies whether an object can exist independently, or whether it must be related to existing objects from the other class.

We first analyse the cardinality property, and within each cardinality case, we will analyse the dependency property. In the ensuing discussion, we use A and B to denote two object classes, and A:B to denote a relationship between the two classes.

Cardinalities are classified into three types. A relationship A:B of cardinality 1:1 restricts each object of class A to be related to at most one object of class B, and vice versa. A relationship A:B of cardinality 1:N restricts each object of class B to be related to at most one object of class A, but places no restrictions on the number of objects of class B that are related to a single object of class A. Finally, a relationship A:B of cardinality M:N places no restrictions on the number of objects of either class that are related to an object of the other class.

Dependencies are classified into four types. A relationship A:B can be a total depen' ncy, a partial dependency A on B, a partial dependency B on A, or a no-dependency. A total dependency requires each class A object and each class B object to be related to a specified number of objects of the other class. The specified number of objects is restricted by the cardinality of A:B. A partial dependency of A on B requires each object of class A to be related to a specified number of objects of class B. Again, the number is restricted by the cardinality of the relationship. Partial dependency of B on A is defined symmetrically. Finally, a no-dependency relationship places no restrictions.

We now discuss the different dependency properties as they apply to a relationship of given cardinality.

2.2.2.1 *One-to-one relationships*

Consider a relationship A:B of cardinality 1:1. We can distinguish four dependency cases.

(a) Total dependency:

Every object in class A must be related to one object of class B, and vice versa. In this case, a one-to-one correspondence exists between the objects of the two classes.

(b) Partial dependency of A on B:

Every object in class A must be related to an object of class B, but some objects in class B can exist that are not related to an object from class A. Mathematically, this relationship is defined by a total 1:1 function from A into B. The inverse is a partial 1:1 function from B onto A.

(c) Partial dependency of B on A:

Symmetric to (b).

(d) No-dependency:

Objects can exist in either class that are unrelated to an object of the other class. This relationship is defined by two partial 1:1 into functions that are inverses of one another.

These properties are *relevant in the* definition of the semantics of one-to-one relationships. We give examples to illustrate this.

First consider a marriage relationship in a monogamous *society between two object* classes HUSBANDS and WIVES. Here, every husband is related to exactly one wife, and vice versa. When a couple gets married, a husband is added to the object class HUSBANDS, a *wife* to the object class WIVES, and the two objects are related together. When a divorce takes place, one husband and one wife object are removed from their respective object classes. This relationship is a total dependency of cardinality 1:1.

Next, *consider the relationship between* DEPARTMENT-MANAGERS and DEPARTMENTS in a company where policy is that a manager can only manage one department. Short periods of time may exist when a *department* has no manager (if a manager resigns suddenly and a search is needed for a *successor*). Here, every manager must be related to a department *object*, but at times a *department may* exist that is not related to a manager. This relationship is a partial dependency of *DEPARTMENT-MANAGERS on DEPARTMENTS of* cardinality 1:1.

Finally, consider the relationship CAPTAINS:MERCHANT-SHIPS that relates a ship to its current captain in a ships-monitoring situation. At a given moment, some ships may not be operational, and hence no captain is assigned to them. Similarily, some captains may be on leave, and unassigned to a ship. This relationship is a no-dependency of cardinality 1:1.

2.2.2.2 *One-to-N relationships*

Consider a relationship A:B of cardinality 1:N. Here, an object of class B can be related to at most one object of class A, but an object of class A can be related to any number of class B objects. If N is specified as a number, this restricts the number of class B objects related to a single class A object to a *maximum of* N. We can again distinguish four types of existence dependencies for such a relationship.

(a) Total dependency (i):

Every object of class A must be related to at least i objects, $i > 0$, of class B, and every object of class B must be related to exactly *one object of class* A. *The relationship* is defined by a total function from B onto A.

(b) Partial dependency (i) of A on B:

Every object of class A must be related to at least i objects, $i > 0$, of class B. An object of class B may be related to at most one object of class A. The relationship is defined by a partial function from B onto A.

Note that we define the dependency of class A objects *on* class B objects for a 1:N relationship A:B by requiring an *object* of class A to be related to at least i objects of class B, $i > 0$. We do not define it by requiring an object of class A to be related to exactly i objects. The latter case is defined by a dependency i if we also specify $N = i$. We then have (minimum i, maximum i) or exactly i *objects of class* B related to each object of class A.

(c) Partial dependency of B on A:

Every object of class B must be related to exactly one object of class A. The relationship is defined by a total function from B into A.

Here, i need not be specified, since it is restricted to 1 by the cardinality.

(d) No-dependency:

Objects can exist in either class that are unrelated to objects of the other class. The relationship is defined by a partial function from B into A.

We can illustrate the four dependency cases of a 1:N cardinality by semantic variations on a 1:N relationship between the two classes DEPARTMENTS and EMPLOYEES. The 1:N cardinality specifies the rule that an employee object can be related to at most one department object. Additional semantics of the relationship in the real-world situation define the dependency properties of the relationship.

First consider the case where every department must have at least i employees. If in addition every employee must be assigned to a department, the relationship is a total dependency (i) (every employee object must be related to a department object, and every department object must be related to at least i employee objects). Alternatively, if new employees are allowed to exist without being assigned to a department, the relationship is a partial dependency (i) of DEPARTMENTS on EMPLOYEES (every department object must still be related to a minimum of i employees but employees can exist that are not related to a department).

Next consider the case where many small departments exist, and the department structure is not stable so that new departments are created and old ones replaced. Here new departments are allowed to exist that do not have any employees. If every employee must be assigned to a department, we have a partial dependency of EMPLOYEES on DEPARTMENTS (*departments may exist that are not related to any employee object*, but every employee must be related to a department). Alternatively, if new employees are allowed to exist that are not assigned to a department, we have a no-dependency.

2.2.2.3 *M-to-N relationships*

Finally consider a relationship A:B of cardinality M:N. This is the most general cardinality. No restrictions exist on the number of objects related to an object of either class. If M or N or both are specified by numbers, a restriction on the maximum number of objects related to a single object of the other class is specified as in the 1:N case. We can again distinguish four types of existence dependencies for such a relationship.

(a) Total dependency (i,j):

An object of class A must be related to at least i objects of class B, $i > 0$, and an object of class B must be related to at least j objects of class A, $j > 0$.

(b) Partial dependency (i) of A on B:

An object of class A must be related to at least i objects of class B, $i > 0$.

(c) Partial dependency (j) of B on A:

An object of class B must be related to at least j objects of class A, $j > 0$. This is symmetric to (b).

(d) No-dependency:

No restrictions exist on the relationship. This is the most general case with absolutely no restrictions.

These M:N relationships are not functional in the mathematical sense.

A total dependency (i,j) makes sense for a relationship A:B of cardinality M:N only if at least i objects of class B, and at least j objects of class A are initially created, since each object from class A must be related to at least i objects of class B, and each object from class B must be related to at least j objects of class A.

An example of a partial dependency (i) of cardinality M:N is the relationship between two classes BILLS-PASSED and CONGRESS-PERSONS in some legislature situation which relates each passed bill with the congress-persons who voted yes on the bill. If we assume that a particular congress has 100 members, and a passed bill requires at least 51 yes votes, the relationship BILLS-PASSED:CONGRESS-PERSONS is a partial dependency of BILLS-PASSED on CONGRESS-PERSONS of cardinality M:N, with $i = 51$.

An example of a partial dependency (1) is the CAN-SUPPLY relationship which relates suppliers to the parts they can supply in the company situation. A reasonable rule to have is that a supplier may not exist in the situation unless he can supply at least one part the company uses (otherwise what good does he serve). The relationship is a partial dependency (1) of SUPPLIERS on PARTS of cardinality M:N.

An example of a no-dependency M:N relationship is the CURRENT-SUPPLY relationship between SUPPLIERS and PARTS in a company situation. Suppliers can exist that do not supply any parts at some moment, and parts may exist which are not currently being supplied. The relationship relates each supplier with the parts he currently supplies.

2.2.2.4 *Specifying the cardinality and dependency by restricting relating properties*

As we discussed in Section 2.1.1, relationships can be specified by inclusion of relating properties in the object classes that participate in the relationship. This is appropriate for binary relationships, but becomes difficult to specify in relationships between more than two object classes.

By appropriate characterization of the relating property, different cardinality and dependency constraints can be specified. Table 2 shows the correspondence. In this table, we assume a relationship A:B with the relating property specified as a property of class A.

In this table, the relating properties that specify the relationship exist in only one of the object classes that participate in the relationship. The structural properties (cardinality and dependency) can then be completely and unambiguously specified.

Relating property	Cardinality	Relating property	Dependency
non-repeating	N:1	mandatory	partial A on B
non-repeating, unique	1:1	non-mandatory	no-dependency
repeating	M:N		
repeating, unique	1:N		

Table 2 Specifying structural properties by restricting the relating property

If the relating properties are specified twice, once in each of the object classes that participate in the relationship, the structural properties must be the same when specified on both sets of relating attributes. Otherwise, the relationship is specified differently on both occassions, and hence an inconsistency exists. For this reason, and because of the non-symmetric specification of relationships when specified via relating properties, it is advantageous to specify relationships by relationship classes.

2.2.2.5 *Summary and a different classification*

In summary, two structural properties of binary relationships A:B were specified: the cardinality and the dependency. Three cardinalities are distinguished, 1:1, 1:N, and M:N. Four types of dependencies were identified: total, partial A on B, partial B on A, and no-dependency. We find $3 \times 4 = 12$ distinct relationships, of which two pairs (partial dependency A on B, and B on A for 1:1 and M:N) are symmetric. Hence, we are left with 10 non-symmetric cases.

Abrial [Abr74] has given a different classification for binary relationships. For a relationship A:B, he defines the structural properties by two number ranges $(a_1 – a_2)$ and $(b_1 – b_2)$. The numbers $(a_1 – a_2)$ mean that a_1 is the minimum number of objects and a_2 the maximum number of objects of class B that can be related to a single object of class A. The meaning of b_1 and b_2 is the same as that of a_1 and a_2 if we reverse the roles of A and B.

Table 3 shows the correspondence between Abrial's numbers, and our classification. In this table, we allow the specification of M and N as numbers if a maximum is known for the cardinality. If no maximum is known, M and N become infinity.

Although Abrial's classification is more concise, we believe that our classification is closer to the way people think about relationships, which is the objective of the concepts discussed in this chapter. Abrial's classification may be more appropriate, for example, if the objective is to implement a system that utilises these concepts. In particular, a clear distinction, as made by us, between the cardinality and the dependency is closer to the semantics of real-world relationships, and easier to understand than pairs of numbers. We will also show in Chapter 5 that it is straightforward to choose a representation for the relationship in a data model if the two structural properties are distinguished, and in Chapter 6 we use our classification when we discuss data model integration. As we can see from Table 3, both classifications are equivalent.

		Range of cardinalities of A:B		
Dependency	Abrial	1:1	1:N	M:N
No dependency	$(a_1–a_2)$ $(b_1–b_2)$	(0–1) (0–1)	(0–N) (0–1)	(0–N) (0–M)
Partial A on B	$(a_1–a_2)$ $(b_1–b_2)$	(1–1) (0–1)	(i–N) (0–1)	(i–N) (0–M)
Partial B on A	$(a_1–a_2)$ $(b_1–b_2)$	(0–1) (1–1)	(0–N) (1–1)	(0–N) (j–M)
Total dependency	$(a_1–a_2)$ $(b_1–b_2)$	(1–1) (1–1)	(i–N) (1–1)	(i–N) (j–M)

Table 3 Correspondence between the two different classifications

2.2.3 *Relationships between categories*

So far, we have discussed relationships between two or more object classes, such that a binary relationship is defined on two object classes, a ternary on three, and an n-ary relationship on n object classes. The general case occurs when the relationships are between *categories* of object classes, rather than object classes. The concepts of categories and roles (see below) which we discuss here are from [BaDa77] and [WeWo80]. They were also discussed in [Kent79].

A *category* of object classes is a group of distinct object classes that participate on the same side of a relationship. In all our previous discussion, only one object class belonged to a *category*. Each category plays a *role* in the relationship. A *role* is a name which signifies the meaning a category plays in the relationship.

For example, the CAN-SUPPLY relationship was between two categories, one on each side of the relationship. The first category consisted of the single object class SUPPLIERS and the second category of the single object class PARTS. The role of the first category can be called CAN-SUPPLY and the role of the second category can be called CAN-BE-SUPPLIED-BY.

The general case is when a category is composed of more than one object class that play the same role in a relationship. The canonical example [WeWo80] is an ASSIGNMENT relationship in a company between the company owned vehicles, and the company entities to which each vehicle is assigned. Vehicles can be cars, trucks, or possibly even aeroplanes and ships. Each of these object classes plays the same role in the ASSIGNMENT relationship: that of ASSIGNED-TO.

The other role is that of ASSIGNED-VEHICLES. If we assume that a vehicle can be assigned to a *department*, an individual employee, or a customer, then the object classes that play this role are DEPARTMENTS, EMPLOYEES, and CUSTOMERS. Hence, this ASSIGNMENT relationship is between two categories. The first category consists of the four object classes CARS, TRUCKS, AEROPLANES, and SHIPS, and the second category consists of the object classes DEPARTMENTS, EMPLOYEES, and CUSTOMERS.

We will further discuss categories in Section 5.4.3, when we examine how they can be represented in the structural model.

2.3 CONCLUSIONS

In this chapter, we presented the concepts we consider important in modelling the semantics of real-world situations. Many of these concepts have been discussed in the literature as indicated in each section. The characterisation of structural properties of relationships is our own.

In summary, an object class O is a class of objects $\{o_1, o_2, \ldots\}$ of identical structure that correspond to real-world objects which are described using similar properties. Objects in the class O are created and destroyed to correspond to the objects of the class in the real-world.

An object class O is specified by a set of properties $\{p_1, \ldots, p_n\}$, and each property p_i is associated with a set of values $\{v_{i_1}, v_{i_2}, \ldots\}$.

An object is formed by data items from the value sets of the properties. Properties are characterised as shown in Table 1, which constrains the values a property can take for each object in the class as discussed in Section 2.1.1.

Relationships are *mappings* between object classes, or categories of object class. Relationships are described either by relating properties of a class, or by relationship object classes as discussed in Section 2.2. Relationships are constrained by their structural properties, which restrict the possible mappings of objects. Structural properties are discussed in Section 2.2.2.

In the next chapter, we define the Structural Model, and in Chapter 4, we present the Structural Model Query Language, which is object-oriented, and hence its statements are close to the concepts discussed here. In Chapter 5, we show how the Structural Model represents the concepts discussed in this chapter.

3 THE STRUCTURAL MODEL

3.1 INTRODUCTION

The descriptions of real-world situations given in Chapter 2 cover many concepts that are used in data modelling. For the design of a generalised database management system, it must be possible to define these concepts in a formal model. In this chapter, we present the *Structural Database Model*, which is a formal tool for data model design. The structural model is also a formal tool to base upon a generalised database management system.

In this thesis, we consider the use of the structural model for data model design (Chapter 5) and data model integration (Chapter 6). We also examine the support that the structural model provides for a user interface (Chapter 4). The use of the structural model for database management system implementation is not discussed in detail, and should be examined in future work.

Database models are used as a formal means of describing real-world situations about which data is to be managed, and as a basis for implementing the generalised database management system. To be able to satisfy both requirements—represent reality and serve as a basis for implementation—the models are based on data structural concepts rather than real-world concepts. Hence, the models we are concerned with represent the *information structure* of the database, rather than the real-world. However, as we shall see in Chapter 5, a few well chosen data structures can go a long way towards representing reality.

A database model should be convenient to use for retrieval and update of the database by the users by providing a high-level user interface that is close to the users perceptions. It should also make implementing the processes that handle the update and retrieval on the computer system feasible.

In Section 1.2, we gave five criteria which we believe are important for the choice of a formal database model. Here, we expand on these points.

(1) Correctness: The model can be used to describe many of the semantic concepts needed for data modelling correctly. We discussed many of these concepts in Chapter 2. In most cases, rules are known about the situation that need to be maintained, such as the characteristics of properties, and the structural properties of relationships. The model should provide a framework for declaring these rules whenever they are known. Then, the processes that use the model are aware of the rules, and can utilise them automatically, whether to enforce constraints on the database or to respond to users queries about the structure of the data.

(2) Simplicity: The model should not be complex so that it becomes difficult and awkward to use, both for modelling and for database implementation. Hence, it is important to search for a small set of underlying structures that can effectively be used to model the frequently encountered situations. These structures need not be identical to the semantic concepts. Rather, they should be able to represent the semantic concepts correctly, as well as to provide the ability to support processes for user interaction that are close to the user's perception of the situation.

(3) Formality: The model should be based on formal mathematical principles, since this allows processes that use the model to be formally specified and easier to verify. This also allows verification of the capabilities of the model based on developed theoretical concepts.

(4) Implementation feasibility: An efficient implementation of the model should be feasible, since one of the reasons for the model is to create a database management system based upon the model. That is why real-world models are not used at this level. The structures developed in the model should lend themselves to automatic translation to storage structures.

(5) Data independence: Though point (4) is very important, it is just as important to separate the data model from the implementation, since both modelling and implementation are quite complex. This also allows restructuring of implemented access paths when the usage patterns of the database change without affecting the data model. Hence, user processes based on the model need not be changed when the access structures change. Restructuring is expected to be important in future database systems, hence a clean interface should exist between the data model and the implementation.

The structural model which we use in this thesis is based upon the relational model [Codd70], augmented with the concept of connections between relations. The connections are used to represent structural properties of relationships as well as to represent subclass concepts, which have been discussed in Chapter 2.

The main contribution of the structural model is that structural properties of relationships are represented in the model itself, rather than being attached to the model. The connections between relations represent constraints that are capable of defining the structural properties of relationships. This allows implementors to consider efficient methods of checking these constraints when the file and access structures are designed. This is not possible if these constraints are defined separately from the model, either in a general integrity assertion subsystem (as in relational systems) or procedurally by attaching programs to check these constraints (as in network systems).

The main point is that if integrity constraints exist in the real-world situation, and if it is important that the database maintain these constraints, then the constraints must be identified during the model design process, and taken into account in an implementation. These constraints are properties of the real-world situation as much as the data items in the database, and are central to the data model design process.

The other important contribution of the model is that *many semantic concepts* can be *represented* correctly using a small set of data structural concepts. As we shall see in Chapter 5, the structural model can correctly represent all the semantic concepts discussed in Chapter 2, even though the only concepts in the model are those of domains, *attributes, relations, and connections. By careful* specification of different characteristics of attributes, and the choice of the types of connections, the structural constraints are specified.

There are other types of constraints, which we call *domain* constraints. Again, if these are rules of the real-world situation, they should be specified with the model. These constraints cannot be specified as part of the structural model, but should be specified on the model. These constraints should also be taken into account when designing the implementation.

Another important aspect of the structural model is that the additional information represented in a user data model serves to identify differences of view among the different user groups that are expected to use the database. This leads to the development of the data model integration methodology, which we discuss in Chapter 6. The integration allows a three-level database management system architecture as we discussed in Section 1.3, similar to the ANSI/X3/SPARC architecture.

A major difference exists between the conceptual schema of ANSI/SPARC (see *Figure 1*), and our *integrated database model. In the* ANSI/SPARC approach, the conceptual schema is designed first, and then different user groups can define their external schemas based on the conceptual schema. Hence, they are restricted by the database administrator's decisions of what is represented in the conceptual schema, and what is *not. This is the premise of* current database systems that allow multiple user schemas or views, such as the subschema mechanism in CODASYL network systems, or the view mechanism in relational systems. Hence, this type of system is inherently centralised.

In the approach we take, the different users are the ones who define their data models, based on their expected needs and *requirements. Then,* a data model integration phase follows which produces the integrated database model. Hence, the integrated database model includes information about the different classes of data that different users include in their models, as well as to common data. This allows us to integrate *different user data models to form* an integrated database model which correctly supports the different user models, including their structural constraints, as well as giving update capability to the users, which is not always possible in the relational view mechanism.

Another aspect of the explicit representation of connections, other than the expression of structural integrity constraints, is their use in the structural model query language (see Chapter 4). By giving names to connections in two directions, it is possible to omit the specification of the JOIN operation, which relates data from two relations [see Codd72a], if the JOIN is based on defined connections. Another feature of the

language is that the comparison operators compare values as well as sets, which leads to straightforward expression of queries that require quantification in a predicate calculus based language.

In the next section, we formally define the structural model with a brief discussion of how the different concepts are used to model real-world structures. In Chapter 4, we present a user interface to the structural model, and in Chapter 5 we give a more detailed discussion of how the semantic concepts discussed in Chapter 2 can be represented, as well as the use of the structural model in data model design.

3.2 THE STRUCTURAL DATABASE MODEL

The structural model is based upon the model which is described informally in Chapter 7 of [Wied77], which in turn is based on the relational model. Wiederhold's model categorizes relations into five types and states rules for maintenance of structural integrity based upon the interaction between types of relations.

This model is augmented here by introducing the concept of connections between relations, and a method for representing subrelations based on connections.

We now give a formal definition of the structural model. We start by the definition of relations, then define connections between relations. Relational concepts are well known. However, for completeness we will concisely define relations and relation schemas as we use them in the structural model.

In order to define a relation, we will define attributes, relation schemas, and tuples. We use B, C, D, to denote single attributes; X, Y, Z, to denote sets of attributes; *b*, *c*, *d*, to denote values of single attributes; and, *x*, *y*, *s*, to denote tuples of values for sets of attributes. For simplicity, we will assume that all sets of attributes are ordered.

3.2.1 Relations

Definition 1: An ***attribute*** B is a name associated with a set of values, *DOM*(B). Hence, a ***value*** *b* of attribute B is an element of *DOM*(B).

Note that in our definition of attribute, both the name and the domain define the attribute. Hence, *two different attributes* may have the same name but different domains. However, within a single relation schema (see Definition 3), all attribute names must be unique.

For an (ordered) set of attributes $Y = \langle B_1, \ldots, B_m \rangle$, we will write *DOM*(Y) to denote $DOM(B_1) \times \ldots \times DOM(B_m)$, where $\times$ is the cross product operation. Hence, *DOM*(Y) is the set $\{\langle b_1, \ldots, b_m \rangle \mid b_i \in DOM(B_i) \text{ for } i = 1, \ldots, m\}$.

Definition 2: A ***tuple*** y of a set of attributes $Y = \langle B_1, \ldots, B_m \rangle$, is an element of *DOM*(Y).

Definition 3: A ***relation schema*** of order m, $m > 0$, is a name associated with an (ordered) set of attributes $Y = \langle B_1, \ldots, B_m \rangle$. A ***relation***, R, defined by the relation schema Y is a current instance (or value) of relation schema Y, and is a subset of *DOM*(Y).

Each attribute in the set Y is required to have a unique name.

The set Y is partitioned into two subsets, K and G. The ***ruling part***, K, of relation schema Y is a set of attributes $K = \langle B_1, \ldots, B_k \rangle$, $k \leq m$, such that every tuple y in R has a unique value for the subtuple corresponding to the attribute set K for all relation instances R of Y. For simplicity, we let K be the first k attributes of Y. The ***dependent part***, G, of relation schema Y is the set of attributes $G = Y - K$, where $-$ is the set difference operator.

We will write R[Y] or $R[B_1, \ldots, B_m]$ to denote that relation R is defined by the relation schema $Y = \langle B_1, \ldots, B_m \rangle$.

Also, K(Y) will denote the ruling part of relation schema Y, and G(Y) will denote the dependent part. Similarly, for a tuple y in relation R, defined by the relation schema Y, $k(y)$ will denote the subtuple that corresponds to the attributes K(Y), and $g(y)$ will denote the subtuple that correspond to G(Y).

A relation R[Y] may have several attribute subsets Z of Y which satisfy the uniqueness requirement for ruling part. In the structural model, the ruling part of a relation schema is defined according to the type of the relation (see Section 3.2.3).

The tuples in a relation are not static. Whenever a relation is updated, attribute values are changed within tuples. Tuples are also constantly being inserted into and deleted from a relation. These insert, delete, and update operations create new relation instances which are specified by the same relation schema, and reflect changes in the real-world situation that the relations represent.

Not all relation instances of a particular relation schema are meaningful. Constraints must be specified on a relation schema to ensure that relation instances of the schema correspond to meaningful objects. One such constraint is specified by the ruling part, which restricts relation instances to those which have unique values for the ruling part attributes in all tuples of the relation.

Other constraints are specified on the attributes of a relation schema, as we discuss below. These ***attribute*** constraints are specified on individual relations. Some constraints are specified between relations, which we call ***inter-relation*** constraints. Inter-relation constraints are specified by connections as discussed in Section 3.3.1.

Definition 3a: A set of attributes Z in relation schema Y may be grouped together and given a name. The domain of this *compound* attribute is the cross-product of the individual domains of the attributes that form the set.

Definition 3b: (Attribute Constraints) An attribute B in a relation schema Y may be specified to be:

(a) *Unique*: no two tuples in any relation instance of schema Y have the same value for attribute B.

(b) *Optional*: values of *none* or *unknown* are allowed for attribute B in tuples of relation instances of Y.

(c) *Derivable*: the value of attribute B is calculated from other values in the database. Here, the method of calculating the value for tuples in the relation must be specified with the attribute.

The attribute constraints of Definition 3(b) apply to both compound and simple *attributes.*

When no additional constraints are specified on attributes, the default specification for an attribute are *non-unique, mandatory* (*no none or unknown* values, or *null* values, allowed), and actual (values not derivable). Attributes that allow null values must be explicitly specified as optional. *This is because the main reason for null values is the* inclusion of many attributes in a relation that are applicable to only subsets of tuples of that relation. The original relational model did not have subclass constructs, which can be used to handle the majority of these cases.

Repeating attributes are not allowed *in relations, since they violate first normal* form. Repeating attributes are represented by nest relations as we shall discuss in Section 3.2.3. Relating attributes, which specify relationships, are specified by the connections between relations, as we discuss in Section 3.2.2.

An example of a relation schema EMPLOYEE that represents the non-repeating properties of the object class EMPLOYEES is shown in Figure 8. Names of compound attributes are *marked* (*, *and the compound attribute is formed from the group of* attributes up to the closing parentheses.

relation schema name: EMPLOYEE

ATTRIBUTE NAME	ADDITIONAL CONSTRAINT
NUMBER	unique
(*NAME:	unique
FIRST-NAME	
MIDDLE-NAME	optional
LAST-NAME)	
(*ADDRESS:	
STREET	
NUMBER	
APARTMENT-NUMBER	optional
CITY	
STATE	
ZIP-CODE)	
(*BIRTH-DATE:	
MONTH	
DAY	
YEAR)	
AGE	derivable from BIRTH-DATE
SEX	
SALARY	
JOB	
DEPARTMENT	
SUPERVISOR	

Figure 8 The relation schema EMPLOYEE and its non-repeating attributes

In Figure 8, the relation EMPLOYEE represents an object class of employees *in some company. A correspondence exists between tuples in the current relation in*stance, and the employees who currently are known to work for the company. The unique attribute NUMBER of EMPLOYEE is the ruling part in this case, and defines the correspondence between an employee with a given number and the tuple with the same value for the NUMBER attribute. The uniqueness constraint, which is implicit if NUMBER is specified as ruling part, ensures that only one tuple will have this number *in a relation instance.*

3.2.2 Connections

We now define the concept of a connection between two relation schemas. Three types of connections are used: direct reference, identity reference, and ownership connection.

Definition 4: A *connection* from relation schema X_1 to relation schema X_2 is an (optional) name, and two sets of attributes Y_1 and Y_2 such that:

(a) $Y_1 \subseteq X_1$.

(b) $Y_2 \subseteq X_2$.

(c) $DOM(Y_1) = DOM(Y_2)$.

(d) Y_1 and Y_2 cannot be optional.

We then say that X_1 *is connected to* X_2 *through* (Y_1, Y_2), and Y_1 and Y_2 are called *connecting* attributes.

A connection is hence an ordered pair (X_1, X_2) and can be represented by a directed arc from X_1 to X_2. The connection is fully specified by the relation schemas X_1 and X_2, *and the connecting attributes* Y_1 *and* Y_2.

The connection name is used for data access purposes only, to specify retrieval and update requests in the Structural Model Query Language (see Chapter 4). A connection with *no name cannot be used in such requests*, and only serves to define inter-relation constraints.

Definition 4a: Two tuples x_1 and x_2 from relation instances defined by the two schemas X_1 and X_2 respectively are *connected* if $y_1 = y_2$, where y_1 and y_2 are the *subtuples of the tuples* x_1 *and* x_2 *that correspond to the connecting attributes* Y_1 and Y_2.

Definition 4b: The name of a connection is optional. If it is defined, it can be a single name of the form (*name*$_1$), or two names of the form (*name*$_1$–*name*$_2$), where *name*$_1$ refers to the direction X_1 to X_2, and *name*$_2$ refers to the reverse direction.

Sometimes, *derived* connections (see Chapter 4) are needed to provide retrieval paths. Derived connections must have a name but do not specify any inter-relation constraints.

Connections may be more complex than as defined above. For example, if we desire a connection between two sets of attributes with dissimilar, but related, domains, condition (c) of Definition 4 may by changed to:

$$DOM(f(DOM(Y_1))) = DOM(g(DOM(Y_2))).$$

Here, f and g are arbitrary functions that have the same domain for their ranges. This would allow us to relate arbitrary domains when the above function are defined. We do not use these generalised connections in this thesis.

Three types of connections exist in the structural model—direct reference, identity reference, and ownership connections. We define them here, and then briefly discuss their use in modelling.

Definition 5: A *reference connection* from relation schema X_1 to relation schema X_2 through (Y_1, Y_2) is a connection from X_1 to X_2 through (Y_1, Y_2) such that:

(a) $Y_2 = K(X_2)$.

(b) At all times, if two relation instances $R[X_1]$ and $R[X_2]$ exist simultaneously, all tuples in $R[X_1]$ must be connected to existing tuples in $R[X_2]$.

We then say that X_1 *references* X_2, and X_2 is *referenced* by X_1.

Definition 5a: A reference is an *identity reference* if:

(a) $Y_1 = K(X_1)$.

(b) Deletion of a tuple from $R[X_2]$ causes the deletion of any tuple connected to it in $R[X_1]$.

Definition 5b: A reference is a *direct reference* if it is not an identity reference. For a direct reference, deletion of a tuple from $R[X_2]$ that is connected to (referenced by) a tuple in $R[X_1]$ is prohibited.

Identity references are used to represent subrelations of a relation, as we discuss in Section 3.2.3, and the connection is from the subrelation to the relation.

Definition 6: An *ownership connection* from relation schema X_1 to relation schema X_2 through (Y_1,Y_2) is a connection from X_1 to X_2 through (Y_1,Y_2) such that:

(a) $Y_1 = K(X_1)$.

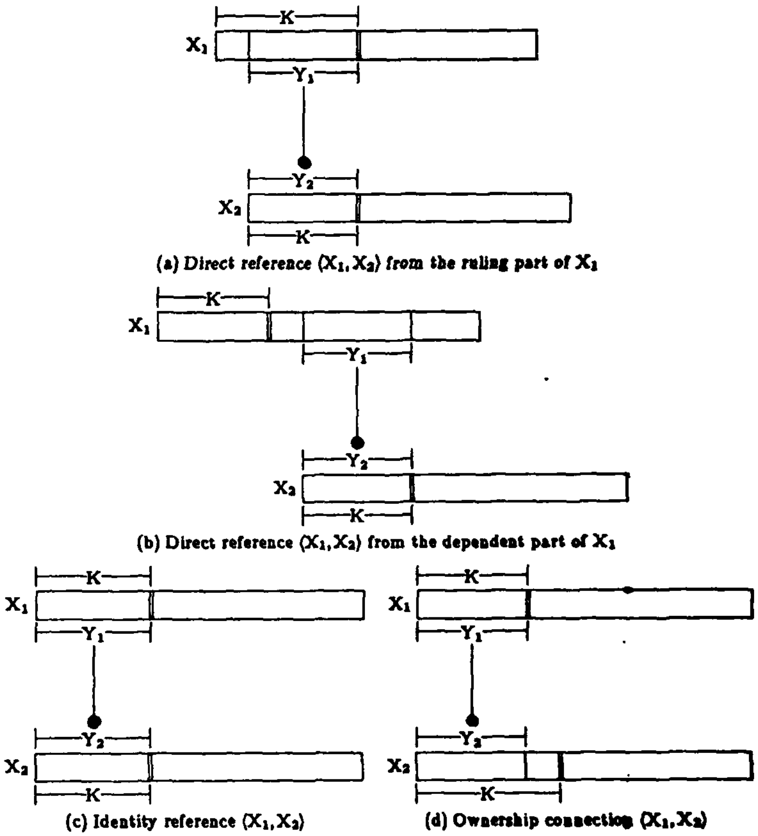
(b) $Y_2 \subseteq K(X_2)$, and $Y_2 \neq K(X_2)$.

(c) At all times, if two relation instances $R[X_1]$ and $R[X_2]$ exist simultaneously, all tuples in $R[X_2]$ must be connected to existing tuples in $R[X_1]$.

(d) Deletion of a tuple from $R[X_1]$ causes the deletion of all tuples connected to it in $R[X_2]$.

We then say that X_1 *owns* X_2, and X_2 *is owned by* X_1.

These connections may be represented graphically as shown in Figure 9. They are represented by directed arcs, with the ● representing the *TO* end of the connection. The ruling part attributes in each relation are marked K, and separated from the dependent part attributes by double lines (‖). The referencing attributes of a direct reference connection can be any subset of the attributes X_1 and are not restricted to the cases shown in Figure 9 (a) and (b).



(a) Direct reference (X_1,X_2) from the ruling part of X_1

(b) Direct reference (X_1,X_2) from the dependent part of X_1

(c) Identity reference (X_1,X_2)

(d) Ownership connection (X_1,X_2)

Figure 9 Types of connections

Lemma 1:

The ownership connection of Definition 6 connects one tuple from a relation instance of X_1 to any number of tuples from a relation instance of X_2. Hence, it is a $1{:}N$ connection from X_1 to X_2. Mathematically, it is a (total) function from $R[X_2]$ into $R[X_1]$.

Proof:

Since the connecting attributes Y_1 of relation schema X_1 are the ruling part, Y_1 is a set of unique attributes. Hence, a value for Y_1 can occur in at most one tuple of a relation instance $R[X_1]$, so that no more than one tuple of $R[X_1]$ can be connected to a given tuple in $R[X_2]$.

Part (c) of Definition 6 ensures that the function is total. QED.

Corollary:

If the connecting attributes Y_2 of X_2 are additionally constrained to be unique, the ownership connection becomes 1:1.

Lemma 2:

The direct reference connection of Definition 5(b) connects a tuple from a relation instance of X_2 to any number of tuples from a relation instance of X_1. Hence, it is an $N{:}1$ connection from X_1 to X_2. Mathematically, it is a function from $R[X_1]$ into $R[X_2]$.

The proof is similar to that of *Lemma 1*.

Corollary:

If the connecting attributes Y_1 of X_1 are additionally constrained to be unique, the direct reference connection becomes 1:1.

The ownership connection and the direct *reference connection* are both $1{:}N$ connections. There are two differences between them:

(1) The maintenance of condition (b) in Definition 5 and condition (c) in Definition 6 is different. For an ownership connection, an owner-owned connection is defined with X_1 being the owner relation and X_2 the owned relation. Hence, when a tuple is deleted from $R[X_1]$, all tuples connected to it in $R[X_2]$ are deleted to maintain condition (c) of Definition 6. The reference connection is different in that if an attempt is made to delete a tuple from $R[X_2]$ which is connected to some tuples in $R[X_1]$ (and hence would lead to a violation of condition (b) of Definition 5), that deletion is not allowed.

(2) The ownership connection is used to provide identification of tuples in the owned relation $R[X_2]$. That is why the connecting attributes Y_2 are always part of the ruling part of X_2.

Lemma 3:

The identity reference connection connects every tuple from a relation instance of X_1 to exactly one tuple from a relation instance of X_2. Hence, it is a 1:1 connection from X_1 to X_2. Mathematically, it is a one-to-one (total) function from $R[X_1]$ into $R[X_2]$.

Proof:

Both connecting attributes Y_1 and Y_2 are ruling parts of X_1 and X_2, and hence are constrained to be unique. Hence, the 1:1 connection. Part (b) of Definition 5 ensures that the function is total. *QED.*

For the identity *reference connection*, condition (b) of Definition 5 is maintained by deleting a tuple from $R[X_1]$ when the tuple connected to it is deleted from $R[X_2]$. This is because the connection is used to *represent* a subrelation $R[X_1]$ of $R[X_2]$ (see section 3.2.3).

Hence we see that connections are not only used to connect tuples, but also define constraints on the two relations that are connected, as well as the method of maintaining these constraints. As we shall see in section 3.3, these constraints can be expressed as simple algorithms to maintain the structural consistency of the database.

In the following section, we briefly discuss the use of relations and connections in modelling. Chapter 4 provides a complete discussion. Relations are categorised into six structural types. These types are not mutually exclusive: the same relation may be of more than one type. Relation types do not correspond directly to the semantic concepts discussed in Chapter 2. Rather, the relation types specify what kind of connections can exist between a relation and other relations. This specifies certain constraints on the relation that ensure the correct representation of semantic concepts, and also aids in the design of an efficient implementation for the structural model.

3.2.3 Types of relations

In this section, we will formally define the different structural types of relations in terms of their connections with other relation types in the data model. These types of relations were proposed by Wiederhold in [Wied77], Chapter 7. We then define subrelations and how a subrelation is connected to its base relation. In Chapter 5, we show how relations and connections are used for data model design. First, we define the terms data model and database.

Definition 7: A *data model* is a collection of relation schemas and connections that represent a real-world situation. A data model that represents a situation does not change unless new object classes are introduced into the situation or new relationships are defined between object classes or new, non-derivable properties of an object class are introduced. We do not expect such changes to occur frequently.

Definition 8: A ***database*** is a collection of relation instances of the relation schemas that define the data model. The tuples in these relations must obey the constraints specified in the data model by attribute constraints and connections between relation schemas. These constraints reflect rules derived from the real world situation. The database changes frequently, reflecting changes to the objects in the real-world situation. These changes are governed by the rules specified in the data model.

The data model can be called the ***intension***, and the database the ***extension*** of a database system.

From now on, we will use the term relation for both relation schema and relation instance, since the meaning will be clear from the context.

There are five relation types in the structural model. These are primary relations, referenced relations, nest relations, association relations, and lexicon relations. We now define each of the relation types.

Definition 9: A ***primary relation*** is a relation that is neither referenced by a direct reference connection nor owned by an ownership connection from any other relation in the data model.

Hence, primary relations cannot have any direct reference or ownership connections to them, so deletion of tuples from primary relations is not directly constrained by the data model.

Definition 10: A ***referenced relation*** X_2 is a relation which has direct reference connections to it from some relations in the data model. A relation X_1 that has such a connection to X_2 is called a ***referencing*** relation of X_2.

The ruling part attributes $K(X_2)$ of a referenced relation are used for referencing X_2. Hence, each relation X_1 that references X_2 will have a set of connecting attributes that define the reference connection to $K(X_2)$.

The reference connection constrains insertion and deletion of tuples in both $R[X_2]$ and $R[X_1]$. Insertion of a tuple x_2 in $R[X_2]$ should precede the insertion of any tuples in $R[X_1]$ that are connected to x_2. Deletion of a tuple x_2 from $R[X_2]$ involves checking that it is not connected to any tuples from any relation $R[X_1]$ that references $R[X_2]$.

Definition 11: A ***nest relation*** is a relation, X_2, which has an ownership connection to it from exactly one other relation, X_1, in the data model. X_1 is called the ***owner*** relation of X_2.

A nest relation X_2 has an ownership connection to it from the owner relation, X_1. Hence, the ruling part $K(X_2)$ consists of two sets of attributes: one set defines the connection with X_1, and an additional set of attributes to uniquely identify tuples in X_2 that are connected to the same owner tuple in X_1.

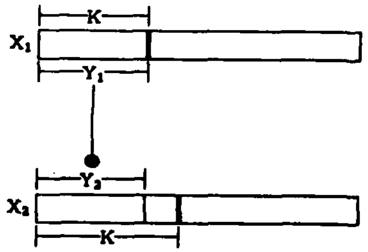


Figure 10 A nest relation, X_2

Insertion of a tuple x_2 in a nest relation $R[X_2]$ requires the prior insertion of its owner tuple in $R[X_1]$—the tuple to which x_2 will be connected. Deletion of tuples from a nest relation $R[X_2]$ may be an explicit deletion, but may also be caused by deletion of the owner tuple from $R[X_1]$, which causes the deletion of all tuples connected to it in the nest relation.

Definition 12: An ***association*** relation of order i, $i > 1$, is a relation X that has i ownership connections to it from i other relations in the data model, $X_1, \ldots X_i$ such that:

(a) Each X_j has an ownership connection to X through (Y_j, Z_j) for $j = 1, \ldots i$.

(b) $K(X) = Z_1 \bigcup \ldots \bigcup Z_i$.

Figure 11 shows an association relation X of order 2 with two owner relations X_1 and X_2. Note that the connecting attributes Z_1 and Z_2 of X do not have to be disjoint as shown in Figure 11. Rather, they may share common attributes.

A tuple in the association is owned by one tuple from each of the owner relations.

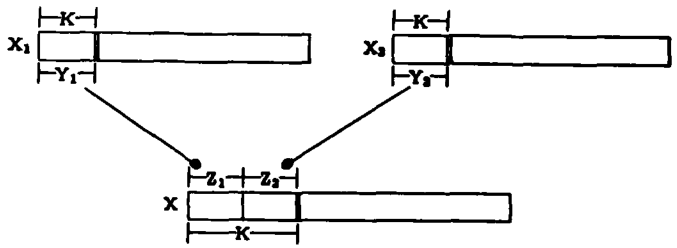


Figure 11 An association relation, X, of order 2

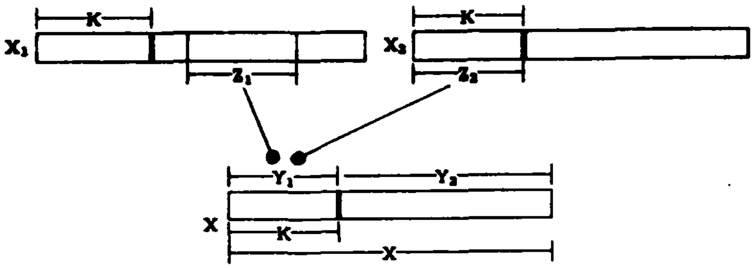


Figure 12 A lexicon relation, X

For each tuple in an owner relation, there may exist sero, one or many owned tuples in the association.

Deleting a tuple from an owner relation causes the deletion of all tuples connected to it in the association. Insertion of a tuple in the association requires the existence of the *i* owner tuples.

Definition 13: A *lexicon* relation X between two sets of attributes Y_1 and Y_2 defines a 1:1 correspondence between *DOM*(Y_1) and *DOM*(Y_2) such that:

(a) $Y_1 \bigcap Y_2 = \emptyset$, and $Y_1 \bigcup Y_2 = X$.

(b) X is referenced by one or more relations in the data model by identity or direct reference connections.

Frequently, object classes can be identified by more that one property. For example, an EMPLOYEES object class may be identified by three properties: NAME, NUMBER and SSN (social security number), and a SHIPS object class may be identified by four properties: NAME, HULL-NUMBER, ID and IRCS (international radio call signal). Lexicons are used to define such 1-to-1 correspondences between attributes. One of the attributes is retained in the relation that represents the object class, and lexicon relations are created to define the correspondence of this attribute to other attributes. This results in relations with a single ruling part, which serves to define all connections with other relations in the model. The lexicon is referenced by this attribute, so the other attributes in the lexicon are accessible.

The above definitions define the five types of relations: primary, referenced, nest, association, and lexicon. Connections can exist at any level in the model: nest relations can be owned by other nest relations, by associations, or by referenced relations as well as by primary relations. A nest or an association may also be referenced. A subrelation may be defined on any relation type.

3.2.4 Subrelations

A subrelation X_S of some relation X defines a subset of the tuples in R[X] as belonging to the subrelation R[X_S]. This subset of tuples has properties that distinguish it from other tuples in R[X]. The relation X is called the *base* relation of the subrelation X_S.

The subrelation has the same ruling part attributes as the base relation, and is connected to the base relation by an identity reference connection. Hence, every tuple in the subrelation R[X_S] is connected to a tuple in the base relation R[X] which has the same value for the ruling part. We do not allow duplication of attributes from the base relation in the subrelation, since the values of these attributes are accessible via the identity connection. I.ence, all attributes (other than the ruling part attributes which specify the connection) of the subrelation have to be different from the attributes of the base relation.

*Definition 14:*A *subrelation* of relation X is a relation X_S such that:

(a) An identity reference exists from X_S to X.

(b) $(X_S - K(X_S)) \bigcap (X - K(X)) = \emptyset$.

The relation X is called the *base* relation of subrelation X_S.

The above definition defines non-restriction subrelations completely. A restriction subrelation must obey additional constraints, which we now define.

Definition 14a: A *restriction subrelation* of a relation X, restricting the set of attributes Y, $Y \subseteq X$, to the subdomain D_S, $D_S \subseteq DOM(Y)$, is a subrelation X_S of X such that: every tuple x in R[X] that has value $y \in D_S$ for the restricting attributes Y is connected to a tuple x_S in R[X_S].

An example of a restriction subrelation is a relation TECHNICAL-EMPLOYEE which is a subrelation of the EMPLOYEE base relation that restricts an attribute JOB of EMPLOYEE to the subdomain of values {*engineer, researcher, technician*}.

Tuples in a restriction subrelation are specified by the tuples in the base relation. In our example, all employee tuples with JOB value *engineer*, *researcher* or *technician* must also exist in the TECHNICAL-EMPLOYEE subrelation, while all other employee tuples cannot exist in this subrelation. (This is the same as the SELECT operator of the relational algebra.) Hence, if a subrelation is specified by a restriction, the tuples in the subrelation are automatically included when the restriction applies.

An example of a non-restriction subrelation is a relation EMPLOYEE-IN-SPECIAL-PROJECT-X. Existence of tuples in this subrelation is specified externally from the database and not by values in tuples of the base relation. It is true that we could introduce an additional two-valued attribute to indicate whether an employee is in the subset or not. However, we do not want to have to specify for every employee whether he is or is not in the subrelation. In particular, if many such small subrelations existed, too many additional attributes would have to be included in the relation. We limit restriction subrelations to restrictions on already existing attributes.

We will use subrelations to represent three cases:

(1) When a subset of a relation is semantically distinct within the data model. Such a subrelation typically has additional attributes that need to be represented in the model.

(2) When integrity constraints require a subset of tuples of a relation to participate in a connection that other tuples in the relation do not participate in.

(3) When we combine data models to form an integrated database model (see Chapter 6), some data models may represent subsets of relations represented in other data models. This has to be reflected in the integrated database model.

The update rules for the base relation and the subrelation are: when a tuple that belongs to the subset represented by the subrelation is inserted in (deleted from) the base relation, the connected tuple which has the same ruling part value should also be inserted in (deleted from) the subrelation. Also, if an update to a tuple in the base relation results in the removal (addition) of the tuple from (to) the subset, the connected tuple should be deleted from (inserted in) the subrelation. For example, if the job of an employee tuple is changed from engineer to manager, the connected tuple in the TECHNICAL-EMPLOYEES subrelation is automatically deleted.

3.2.5 Extended connections

When we discuss the modelling of real-world structures in Chapter 5, the ownership and direct reference connections as defined in Section 3.2.2 are not sufficient to represent the full structural properties of relationships discussed in Section 2.2. To be able to represent all possibilities, we need to add more information to these connections.

Recall that an ownership connection from relation X_1 to relation X_2 defines a means of connecting tuples from $R[X_1]$ and $R[X_2]$ subject to the constraint that every tuple in $R[X_2]$ is at all times connected to a tuple in $R[X_1]$. The connection is 1:*N* so that a tuple from $R[X_2]$ is connected to exactly one tuple in $R[X_1]$.

Sometimes, yet more information is known about the connection. An example is if every tuple in $R[X_1]$ must always be connected to at least one tuple in $R[X_2]$, or to at most *i* tuples, where *i* is a specified positive integer. To be able to specify these constraints, a pair of numbers (*imin*-*imax*) are optionally attached to the connection, such that

$$0 < imin \leq imax \quad \text{and} \quad imax > 1.$$

To specify $imax = 1$ is redundant, since we can specify a unique connecting attribute in X_2 on an unrestricted connection.

When the values for *imax* and *imin* are not specified, the default is 0 for *imin*, and ∞ for *imax*, which gives us the ownership connection defined in Section 3.2.2.

The same numbers can be attached to a direct reference connection, with the roles

of $R[X_1]$ and R[X2] reversed, since a reference connection from X_1 to X_2 is an *N*:1 connection.

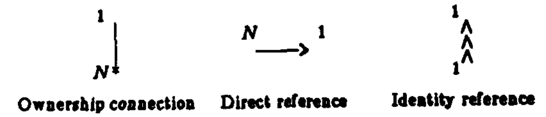
For the identity reference connection, we can only specify a (min 1) since the connection is already specified to be 1:1 . If a (min 1) is specified, a one-to-one correspondence exists between tuples of the two connected relations.

These numbers are used to model 1:*N* and *M*:*N* relationships of different dependencies as we discuss in Section 5.3.

3.2.6 An example

In this section, we give an example to demonstrate the concepts introduced thus far. A more detailed discussion of semantic modelling is given in Chapter 5.

In our diagrams, we will represent the three types of connections as follows:



Whenever a connection is further constrained as discussed in Section 3.2.5, the constraint is attached as a (min *i*, max *j*) to the connection. The name of a relation where applicable is given above the relation schema in the diagram, while the name of the connection is given next to the connection, again where applicable. The precise syntax for a Structural Model Definition Language (SMDL) to be used for definition of such a structural model, and a Structural Model Query Language (SMQL) to be used for retrieval from and update of a structural database are given in Chapter 4.

The example in Figure 13 shows a small part of a structural data model that represents a situation within some company. Three main object classes are represented: EMPLOYEES, DEPARTMENTS, and PROJECTS. These are represented by relations named EMPLOYEE, DEPARTMENT, and PROJECT respectively.

In our diagrams and queries, we will use capital boldface for relation names, lowercase boldface for connection names, and standard capital letters for attribute names.

Note the use of the same name for attributes that describe the employee number, department number, and project number. The three attributes are different since the domain for attribute NUMBER in EMPLOYEE is the set of employee numbers known to the company, while the domains of the attributes NUMBER in DEPARTMENT and PROJECT are the department numbers and project numbers respectively. Attribute names are chosen to make query expression uniform and natural.

To ensure non-ambiguous names, attribute names within a single relation must be unique. Connections specify the connecting attributes, which must have overlapping domains but may have different names.

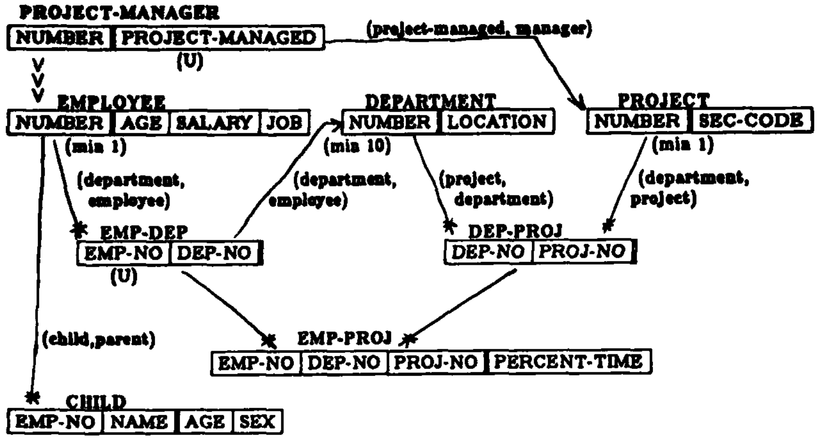


Figure 13 Part of a structural data model

Non-repeating properties of each object class are represented as attributes of the relation. Repeating properties are represented as a nest relation owned by the relation that represents the object class, such as the CHILD relation which represents a repeating property of the EMPLOYEES class. Note that when a tuple is deleted from the relation EMPLOYEE, the tuples owned by it in CHILD are automatically deleted which correctly reflects that it is a property of the employee object.

The association relation *DEP-PROJ serves to connect department and project* tuples. It represents a relationship between the object classes DEPARTMENTS and PROJECTS. This relationship is a partial dependency (1) of PROJECTS on DEPARTMENTS of cardinality $M{:}N$ in the terminology of Section 2.2. These structural properties are derived from an analysis of the situation, so they are rules of the company. In this company, a project must be related to at least one department and this rule is represented by the (min 1) on the ownership connection from PROJECT to DEP-PROJ.

The nest relation EMP-DEP connects an employee tuple with a department tuple. It represents the relationship between EMPLOYEES and DEPARTMENTS which relates each employee object to the department he works in. Here, we assumed the structural properties of this relationship to be a total dependency (10) of DEPARTMENTS on EMPLOYEES, and that the cardinality is $1{:}N$ (*every department must have at least* 10 employees, and an employee works in exactly one department).

The rules to design a data model from the structural properties of a relationship are given in Chapter 5. This example is presented here to briefly illustrate the concepts introduced.

Now consider the relation, attribute and connection names. These are used by the SMQL for retrieval of information from the database. Hence, one can say

Q0: GET (NUMBER, LOCATION) of department of PROJECT
WHERE NUMBER of PROJECT=*11*

The condition NUMBER of PROJECT=*11* selects a particular tuple in the relation named PROJECT. The department property of a project here is not an attribute, but is specified by the connections from PROJECT to DEP-PROJ to DEPARTMENT. Finally, NAME and LOCATION are attributes of the DEPARTMENT relation.

In tracing connections to find an attribute name, connection names that lead from a specified relation are examined. In this example, connection names that lead from PROJECT are examined for the name department. Note that department is the first name of the ownership connection from PROJECT to DEP-PROJ since this is the direction of the connection (from–to). The connection is traced to the relation named DEP-PROJ, where the desired attributes NAME and LOCATION are not found. Hence, a further search for a connection named department from DEP-PROJ is conducted and found since the second name of the ownership connection from DEPARTMENT to DEP-PROJ is department, and so leads in the direction from DEP-PROJ.

The condition for no ambiguity for connection names is that all connection names that lead in the direction *from* a relation are unique. Hence, connection names are not unique within the model, since we may have two connections with exactly the same names that connect different relations. The only condition is that for every relation, all connection names leading out of that relation are distinct. This rule can be easily enforced. The complete SMQL is discussed in Section 4.3.

The association relation EMP-PROJ connects employee tuples with project tuples. It represents a relationship between the object classes EMPLOYEES and PROJECTS which relates employees to the projects they work in. This association is between a nest relation and an association relation. The DEP-NO attribute in EMP-PROJ exists only to represent the constraint that an employee can only work on a project that is associated with his department. The PERCENT-TIME attribute is the percentage of time an employee is assigned to this project.

An additional constraint exists here that the sum of PERCENT-TIME values for a particular employee should not exceed 100. This is an example of a *domain* constraint that cannot be expressed directly in the structural model. This type of constraint is expressed on the model by the *define constraint* statement of the Structural Model Query Language (see Section 4.3).

Now consider the subrelation PROJECT-MANAGER of EMPLOYEE. This represents the subset of employees that are also project managers. An additional attribute PROJECT-MANAGED is needed to describe which project each project manager manages. Here, we assume that each manager manages one project, and vice-versa, so the

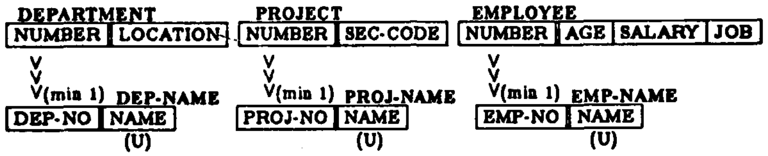


Figure 14 Lexicons used for EMPLOYEE, DEPARTMENT, and PROJECT names

cardinality of the relationship is 1:1. We also assume the relationship is a partial dependency of PROJECT-MANAGERS on PROJECTS, so that every project manager must manage a project. The partial dependency is specified by the direct reference connection from PROJECT-MANAGER to PROJECT, and the 1:1 cardinality is specified by restricting the connecting attribute PROJECT-MANAGED of PROJECT-MANAGER to unique values (specified by a (U) in Figure 13).

Finally we consider the use of lexicon relations. Every department is identified by both a department number and a department name. To simplify the data model, a choice is made to identify a department by its department number. A lexicon relation is created to describe the correspondence between department numbers and names. Note that this is only at the representation level. In an implementation, it is likely that the NAME attribute of DEPARTMENT will be part of the same file as the other attributes that describe a department.

Similar lexicons are created for the EMPLOYEE and PROJECT relations. The connection between a relation and its lexicon may be either an identity connection or a direct reference connection. Whenever the lexicon is of an attribute that is the ruling part of a relation, the connection is an identity connection. To enforce the 1:1 correspondence, a (min 1) is specified on the identity connection. This prohibits having more names than employees, which is allowable if no (min 1) is specified. Figure 14 shows this examples.

Since the ruling part is used in the specification of all three types of connections—ownership, direct reference, and identity reference—the use of a lexicon implicitly specifies the attributes that will be used in all the connection specifications for a relation. In Figure 14, all connections that use the ruling part of EMPLOYEE will refer to the NUMBER attribute. However, for reference by the users, the lexicon attributes—which are the name attributes in this example—are treated like any other attribute. By specifying NAME of EMPLOYEE, the name attribute is accessed just like any other attribute of the EMPLOYEE relation, since all attributes of a base relation (the lexicon in this example) are inherited by the subrelation (see Section 4.3).

3.3 MAINTAINING THE STRUCTURAL INTEGRITY OF THE DATA MODEL

Structural integrity exists in our model when the tuples in the database do not violate the constraints specified on the data model by attribute constraints (constraints within a single relation) and connections constraints (inter-relation constraints). One can consider that the structural model contains a basic set of integrity assertions as part of the data model. We will show in Chapter 5 that these constraints are sufficient to correctly represent all the semantic concepts discussed in Chapter 2.

We do not specify in the data model when or how the integrity constraints on the database are to be maintained in an implementation of the data model. The purpose of the model is that integrity constraints can be recognised, and that implementors can refer for guidance to the model. This will lead to more efficient implementations, since specific implementation structures can be designed to perform the database integrity checks when the database is updated. This is better than defining all constraints in a general integrity subsystem based on propositional logic. Then a large number of constraints will exist in general, and it may be quite expensive to process them.

In current database implementation schemes, periods of time may exist when the structural integrity of the database does not hold. In particular, even if constraints are known to the system, some implementations do not enforce all constraints at update time, but check periodically for violations. An example of such a system is the TOD (Time Oriented Database) system [WiFrWe75]. Also, see the paper by Badal and Popek [BaPo79] for a discussion.

Hierarchical and network database systems tend to require that all integrity constraints be satisfied for those connections that are actually implemented. Pure relational database systems have all constraints expressed as integrity assertion in an integrity subsystem separate from the model, a technique which has proven quite costly.

Our model may appear less powerful than the original relational model since update integrity violations can occur. In the pure relational model, inter-relation connections are not described, but are left to be discovered at query-processing time. The lack of recognition of logical connections between relations in a database model will simplify certain technical problems during update, but does not eliminate semantic inconsistencies relative to knowledge models of the user. Furthermore in many situations it is best to discover and correct integrity violations at the time of update rather than to try and cope with an inconsistent database at query processing time.

In Section 3.3.1, we discuss the integrity constraints that may be specified in the structural model. We then discuss rules for maintenance of the structural integrity of a structural database, and show in Section 3.3.2 how these rules may be expressed as simple algorithms for maintaining the structural integrity upon insertion and deletion of tuples, and update of attribute values.

3.3.1 Update constraints in the structural model

The integrity constraints specified in the structural model are of two types. The first type of constraints are specified within separate relations as constraints on attributes. The second type of constraints are those specified by connections, and are inter-relation constraints.

Attribute constraints specified on a structural model include uniqueness of sets of attributes, and checking that values of attributes are from the allowed domain of values. We will discuss domain definitions with our discussion of the Structural Model Query Language in Section 4.3. The unique attribute constraint specifies that whenever a new value for that attribute is introduced into the database (whether by the update of the attribute value for an existing tuple, or by insertion of a new tuple), this new value must not exist for that attribute in any other tuple of the relation. It is not difficult to implement this when it is known beforehand that such a check is needed.

The constraints specified by the connections are the following:

A *direct reference connection* from relation X_1 to relation X_2 specifies the constraints:

(1) Every tuple in $R[X_1]$ must be connected to a tuple in $R[X_2]$.

(2) Deletion is restricted for tuples in $R[X_2]$. Only tuples that are not connected to tuples in $R[X_1]$ may be deleted. This rule maintains the preceding constraint.

An *ownership connection* from relation X_1 to relation X_2 specifies the constraints:

(1) Every tuple in $R[X_2]$ must be connected to a tuple in $R[X_1]$.

(2) Deletion of a tuple from $R[X_1]$ causes deletion of all tuples connected to it in $R[X_2]$. This rule maintains the preceding constraint.

An *identity reference connection* from a subrelation X_S to its base relation X specifies the constraints:

(1) Every tuple in $R[X_S]$ must be connected to a tuple in R[X].

(2) A tuple in R[X] is connected to at most one tuple in $R[X_S]$.

(3) Deletion of a tuple from R[X] requires deletion of the tuple connected to it from $R[X_S]$.

(4) If $R[X_S]$ is a restriction subrelation, every tuple in R[X] that belongs to the subrelation (specified by the value of the restricting attributes of X) must exist in $R[X_S]$.

Hence, whenever a new tuple is inserted into or deleted from the database, an integrity check should be invoked to verify that the constraints specified by the data model are not violated. It is important to do this at update time because the correction of an erroneous transaction is simpler the sooner it is discovered. If such an erroneous transaction is executed, and it is later discovered that it is not correct, a more complicated procedure has to be invoked to ensure that no subsequent transactions were erroneous because of this transaction. Also, persons who carried such transactions may have to be identified so that they can correct their transaction.

Some transactions involve the insertion or deletion of a set of tuples. In this case, one has to verify that the effect of the complete transaction does not violate the integrity of the database.

In the following section, we give algorithms to check the integrity of the database upon insertion and deletion of tuples, and update of data values within tuples.

3.3.2 Database update algorithms

We give three algorithms for maintaining the structural integrity of the data model by observing the constraints of the connections. The algorithms are described in terms of the connection types defined in Section 3.2.2. The insert, delete, and update algorithms do not take into account constraints specified by min and max constraints (see Section 3.2.5), but can be extended to do so in a straightforward manner.

3.3.2.1 Tuple insertion algorithm

Upon receipt of a request to insert a new tuple x in relation R[X], do the following steps.

(1) Check the consistency of the new tuple with the current tuples in the database:

(1.1) For every relation X_1 that has a reference connection (direct or identity) to it from X, verify that the tuple x_1 that will be connected to x exists in $R[X_1]$.

(1.2) For every relation X_2 that has an ownership connection to X, verify that the tuple x_2 that will be connected to x exists in $R[X_2]$.

(2) If step (a) shows that the new tuple is consistent with the current database, insert it and for every relation X_3 that has an ownership connection from X, send a message to the user reminding him to insert any tuples connected to x in X_3.

Insertion involves two actions: checking that tuples connected with the new tuple exist in the database, and insertion of other tuples connected with the new tuple. The checking can be automatic, but insertion of other new tuples will in most cases be done by the user. For example, the insertion of an employee tuple involves insertion of his children in a nest relation CHILD owned by the EMPLOYEE relation, and of the tuples associating the employee with the department he works for in an EMP-DEP association relation, also owned by EMPLOYEE. However, any new tuples in both CHILD and EMP-DEP are inserted by the user. The system can only remind the user that such data may exist, and if they do exist they should be inserted in the database.

In many cases, a transaction can be defined which inserts tuples in several relations. In the previous example, the employee, his children and department can be inserted simultaneously via an insert employee transaction.

3.3.2.2 *Tuple deletion algorithm*

Upon receipt of a request to delete tuple x from relation R[X], do the following steps.

(1) Check for direct references to x from other tuples in the database: for every relation X_1 that has a direct reference connection to X, check that no tuple x_1 in X_1 is connected to x. If any tuples are connected to x via such direct reference connections, send an error message, and do not complete the deletion.

(2) Check if tuples that will be automatically deleted by the deletion of x may be deleted: for every relation X_1 owned by X (connected by an ownership connection from X to X_1), initiate deletion of the tuples in $R[X_1]$ that are connected to x. For every subrelation X_S of X, initiate deletion of the tuple x_S in $R[X_S]$ that is connected to x.

(3) If all the owned and subrelation tuples of x can be deleted, complete deletion of x and all these tuples. Otherwise, do not complete deletion of x, and send a warning message that x could not be deleted.

Deletion a o consists of two parts: checking that the tuple being deleted is not referenced, and deleting tuples owned by the tuple. The algorithm is recursively applied.

3.3.2.3 *Update algorithm*

Upon receipt of a request to update the value of attribute A in a tuple x, which belongs to relation R[X], do the following steps.

(1) If A is neither an attribute through which X is connected to other relations, nor a member of the ruling part of X, perform the update.

(2) Update of connecting and ruling part attributes:

(2.1) Referencing attributes: if A is an attribute through which X references a relation X_R, check that the new value will reference an existing tuple in $R[X_R]$. If the new value will not reference an existing tuple in $R[X_R]$, do not complete the update and send an error message.

(2.2) Ruling part attributes: if A is a member of the ruling part of X, initiate deletion of x using the deletion algorithm. If deletion can be completed, and insertion of the updated tuple x_N with the new value for A is possible using the insert algorithm, carry out the deletion and insertion. Otherwise, send an error message.

Updating attributes that are not constrained by connections needs no checking, except for possible uniqueness and domain constraints. Only ruling part and referencing attributes can cause an inconsistency to the inter-relation constraints.

4 THE STRUCTURAL MODEL LANGUAGES

4.1 INTRODUCTION

In this chapter, we discuss languages proposed for user interaction with the structural model. These languages provide an interface to the user in a database management system that is based on the structural model. They can also be used as an interface to database systems that are based on other models, although an appropriate mapping has to be provided in this case.

Usually two functions are required from the interface to a database system: the capability to define a data model (and how it is to be stored) to the system, and the capability to retrieve information from, and update the information in, the database system. These two functions are usually separated in large database systems.

The capability to define a data model to the system is called *data definition*, and is usually performed by a *Data Definition Language*, or *DDL*. The capability to retrieve information from, and update the information in, the database is performed by a *Query Language*, or *Data Manipulation Language*.

In current large, multi-user database systems, data model definition is performed only by a *database administrator*, whose function it is to define and maintain the information needs of an organisation. The database administrator is also responsible for granting user groups the authority and resources to use parts of the database of interest to them, and to define the capabilities available to them (retrieval only or update only or both).

In some recent experimental systems, a unified approach is taken. A single language exists both for data definition and data manipulation (for example, SEQUEL [Chea76], the language for SYSTEM-R [Asea76]).

In this chapter, we will define two languages that may be used for the structural model—the Structural Model Definition Language and the Structural Model Query Language. We only present the structure and meaning of the languages, but do not discuss in detail how they would be used in a database management system implementation. The purpose of this chapter is to demonstrate that the structural model can support a high-level query language whose statements are quite close to the real-world concepts of Chapter 2, and yet is formal and unambiguous.

The *Structural Model Definition Language* (SMDL) is used to define the relations, attributes, domains, and connections that specify a structural data model. The structural constraints are implicitly specified by such a structural model.

The SMDL is used only for the definition of a logical database model. Hence, it has no constructs for defining implementation structures. For use in an implementation of a database system, capabilities must be provided—separately or attached to the language—to create file and access structures, and to define the mapping from a data model to its implemented file structures. This is left for future work.

The *Structural Model Query Language* (SMQL) is used to specify retrieval and update operations on the information in the database. Hence, this language will be used continuously during the lifetime of the database. This is in contrast to the SMDL which is used to define the data model, and then used sparingly, only when a data model has to be changed.

We present the SMDL in Section 4.2, and the SMQL in Section 4.3. To specify the syntax of the languages, we use extended BNF (Backus Normal Form). Table 4 gives our notation for BNF.

SYMBOL	USE
::=	derives
<...>	non-terminal symbol
[x]	x is optional (may appear 0 or 1 times)
{x}	closure (x may appear 0 or more times)
x|y	disjunction (x or y)
xy	catenation (x followed by y)
(...)	grouping of alternatives

Table 4 Extended BNF notation used for language definition

Hence, the symbols < > [ ] { } | ( ) are meta symbols of the BNF. When a meta symbol is used in the language definition, it is enclosed between single quote marks. For example '(' specifies an open parentheses.

4.2 THE STRUCTURAL MODEL DEFINITION LANGUAGE (SMDL)

The statements of the Structural Model Definition Language are used to create, delete, and modify structural model constructs. The main statement is the CREATE statement which is used to create attributes, relations, and connections. During the data model design phase, some use is made of the statements that delete and modify constructs until a final design is reached for the data model. Once an implementation is defined, the data model is changed only for two reasons:

(1) When new information that was not defined in the data model, and is not derivable from the information defined in the data model, needs to be represented. This will usually necessitate a physical reorganisation of the database implementation.

(2) When new information derivable from existing information needs to be defined. A derivation specification is all that is needed. No reorganisation of the implementation is necessary, although it may be useful to reorganise the database implementation if the new information is used with great frequency.

The SMDL is defined by the following extended BNF productions. A brief explanation, and additional semantics are given following some of the productions.

```
<SMDL statement>  ::= ( <create statement> | <delete statement>
                       | <modify statement> | <define statement> ) ;
```

For the definition of a data model, only the create statement and the define statement are needed. The delete and modify statements are used only during the data model design process, or when (infrequent) changes are needed.

4.2.1 The CREATE statement

The create statement is used to create domain names, relations, and connections. When a domain is created, it only need be specified by a domain name. The domain name can then be used to specify the domains of attributes of relations.

Creation of a relation involves the specification of its attributes, and the domain for each of the attributes. As mentioned, all domains must be created before their use in attribute specification.

Creation of a connection involves the specification of the connection names, which are optional, as well as the connection type (ownership, direct reference, or identity reference). We also must specify the relation from which the connection originates, and the relation to which it is directed, as well as the connecting attributes in both relations. These two relations must have been previously declared. The syntax for the create statement follows.

```
<create statement>  ::= CREATE ( DOMAIN <domain name>
        | RELATION <relation definition> | CONNECTION <connection definition> )
```

To create a relation, the relation is given a name, which must be unique among all relation names in the schema. The attributes of the relation are defined by giving each attribute a name, and specifying the domain for the attribute. The ruling part is specified, so that the schema processor may check the correctness of connection definitions in which the relation participates. The ruling part attributes are implicitly constrained to be unique (as defined in Section 3.2.1).

```
<relation definition>  ::= <relation name> = <ruling part> : [ <dependent part> ]
<ruling part>  ::= <attribute list>
<dependent part>  ::= <attribute list>
<attribute list>  ::= <constrained attribute> {, <constrained attribute> }
<constrained attribute>  ::= <attribute name> ( COMPOUND '(' <attribute list> ')'
        | DOMAIN <domain name> ) [ INVISIBLE ] [ OPTIONAL ] [ UNIQUE ]
```

Each attribute of the relation must have a name that is unique among all the attribute names of the relation. A compound attribute—which is composed of several attributes (see Section 3.2.1)—is created from a list of other attributes (compound or simple) in the relation, and its domain is implicitly the cross product of the domains of these attributes. Any attribute, whether simple or compound, may be further constrained to be unique, optional, or invisible. (We will see the use of invisible attributes in Section 5.4.3, when we discuss representation of relationships between categories.)

Next we consider connections. A connection is uniquely defined by the two relations it connects, the connecting attributes in each relation, and the type and direction of the connection. The connecting attributes must follow the rules specified by the type of the connection, as defined in Section 3.2.2. The following syntax defines connections.

```
<connection definition>  ::= [ <connection name> ] TYPE <connection type>
        FROM <relation name> '(' <from attributes> ')'
        TO <relation name> '(' <to attributes> ')' [ MIN <number> ] [ MAX <number> ]
<connection name>  ::= '(' <from-to name> [, <to-from name>] ')'
<connection type>  ::= DIRECT [ REFERENCE ] | OWNERSHIP | IDENTITY
<from attributes>  ::= <attribute name list>
<to attributes>  ::= <attribute name list>
<attribute name list>  ::= <attribute name> {, <attribute name>}
<number>  ::= any positive integer
```

The following constraints are enforced on the definition of a connection. The <from attributes> must be attributes of the FROM relation, and the <to attributes> must be attributes of the TO relation. An identity connection cannot have a <number> greater than 1 for MAX and MIN. All names of connections in the direction from a relation must be unique. These include the set of <from-to name>s of connections from the relation, and the set of <to-from name>s of the connections to the relation. This ensures unambiguous SMQL statements that use these connections for retrieval and update.

4.2.2 The DEFINE statement

We now present the define statement. There are four types of define statements. The <define domain statement> is used to specify the value set for a created domain name. The <define derivation statement> is used to define derived relations, attributes of relations, and connections between relations. The <define constraint statement> is used to define non-structural constraints (*domain* constraints) on the data model. Finally, the <define update transaction statement> is used to define a transaction composed of a sequence of primitive update operations.

```
<define statement>  ::= <define domain statement> | <define derivation statement>
        | <define constraint statement> | <define update transaction statement>
```

First consider the <define domain statement>, which is used to associate a set of values with a domain name. The syntax is as follows.

```
<define domain statement>  ::= DEFINE DOMAIN <domain name> TO BE
                                   <domain description>
<domain description>  ::= <value set> | <reference domain>
<value set>  ::= '{'<value> {,<value>}'}'
        | STRING [ALPHABETIC] [<integer>]
        | INTEGER [RANGE<integer>:<integer>]
        | REAL [INTERVAL <real>:<real>]
        | DATE [<date1>][:<date2>]
<reference domain>  ::= <attribute name> OF <relation name> : <low level domain>
<low level domain>  ::= INTEGER | REAL | STRING | DAT3
```

A domain is specified by a set of explicit values, or by *reference* as all the current values that exist in the database for a particular attribute of a particuiar relation. In the latter case, the domain is variable, and a low level domain is specified to enable us to select the operators that apply to that domain. All domain names must be unique. A <value set> can be an explicit set or a range of integers or reals. If no range is specified, it is defaulted to the integer or floating point range of the computer. Alphanumeric and alphabetic strings, possibly of predefined length, are also allowed, as is a type DATE, which has the obvious meaning.

Next, we consider the <define derivation statement>, which is used to define derived structures in a data model. Derived structures are attributes, connections, and relations whose definition is derived from other structures in a data model.

A derived attribute, connection, or relation can be defined at model creation time, or at any time during the operation of the database system, in order to predefine a frequently occuring retrieval concisely. Hence, the statements that define derived structures are part of both the SMDL and the SMQL. We will present these statements with our presentation of the SMQL in Section 4.3.5.

The <define constraint statement> is used to define *domain* or *value* constraints, which cannot be defined by the connections of the structural model. An example of a value constraint is that an employee cannot earn more than his manager, or that a ship cannot carry more cargo than its capacity, say. These types of constraints are defined by logical expressions on relations. All tuples in the relation must satisfy the logical expression. *These constraints are similar to integrity assertions of relational systems* [Ston74].

The <define update transaction statement> is used to define standard update transactions on the database as an attachment to the data model.

The syntax for both the <define constraint statement> and the <define update transaction statement> is given in Section 4.3.5 with the SMQL language definition, since they are also part of the SMQL.

These are the statements used in the definition of a structural model. The domain names are created first, then relations, then connections. Every domain name in a relation definition must be already created. Every relation name in a connection definition must be already created.

Domain value definitions can be defined at any time, but certainly before the development of an implementation for the model. This allows us to define the value set of a domain by reference as the current set of values that exist in the database for a particular attribute of a particular relation, which provides us with a high-level definition capability.

A common class of domains tha: will be defined by reference to attribute values from the database are those domains that describe names of objects, such as EMPLOYEES and DEPARTMENTS. There usually exists a relation where all the objects known to the database *are represented*. The domain of names is then defined by reference to that relation, and the domain will change whenever a new object is added to or deleted from the database. At a lower level, the domain is defined as a string of characters, while at the implementation level, the domain may be defined as a fixed length or variable length string. We should not be concerned with implementation details at the model level.

An important point is that all attributes used to define a particular domain can be recognised, and the data entry for them monitored more carefully and consistently, since the data values for these attributes are critical.

4.2.3 The DELETE and MODIFY statements

Finally, we include the delete and modify statements. These are used during the design phase of the data model, since changes are frequently made to an initial model design as the situation and requirements are better understood and defined more fully. These *statements* are also used to make the infrequent changes to the data model after the database is operational, but such changes are quite expensive since restructuring of the file and access paths may be required. An *exception* is the deletion or modification of derived relations, connections, and attributes, which is not expensive. Hence, a derived structure that is no longer in frequent use can be deleted at any time.

The <delete statement> is used to delete domains, relations, and connections. A domain cannot be deleted unless all relations that include an attribute defined on that domain have been previously deleted. When a relation is deleted, all connections from or to that relation are automatically deleted.

The <modify statement> is used to add attributes to, or delete attributes from, a relation. If any connecting attributes are deleted, the connections defined by the attributes are automatically deleted. This statement is also used to change the domain associated with an attribute of a relation.

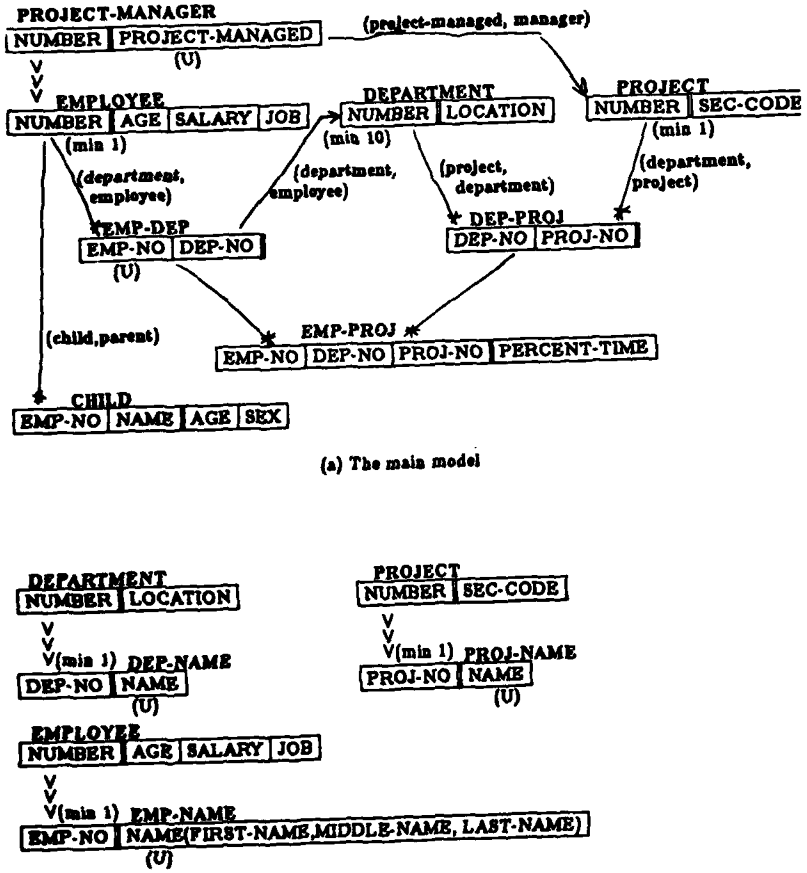
The modify statement is also used to change a domain definition.

```
<delete statement>  ::= DELETE ( DOMAIN <domain name>
       | RELATION <relation name> | CONNECTION <connection definition>)
<modify statement>  ::= <modify relation statement> | <change domain statement>
<modify relation statement>  ::= ADD TO RELATION <relation name> <attribute list>
        | DELETE FROM RELATION <relation name> <attribute name list>
        | CHANGE ATTRIBUTE DOMAIN <attribute name> OF <relation name>
                                                    TO <domain name>
<change domain statement>  ::= CHANGE DOMAIN <domain name> TO
                                                <domain description>
```

4.2.4 An example

We now give an example by defining the model shown in Figures 13 and 14. For convenience, the figures are duplicated here as Figure 15. The attribute NAME of EMP-NAME has been expanded from Figure 14 to a compound attribute.

```
CREATE DOMAIN employee-numbers;
CREATE DOMAIN working-ages;
CREATE DOMAIN salaries;
CREATE DOMAIN job-names;
```



(a) The main model

(b) Lexicons of the model

Figure 15 Part of a structural data model

```
CREATE RELATION EMPLOYEE = NUMBER DOMAIN employee-numbers
        AGE DOMAIN working-ages, SALARY DOMAIN salaries,
        JOB DOMAIN job-names;
DEFINE DOMAIN employee-numbers TO BE NUMBER OF EMPLOYEE : STRING;
DEFINE DOMAIN working-ages TO BE INTEGER RANGE 16:65;
DEFINE DOMAIN salaries TO BE INTEGER RANGE 8000:90000;
DEFINE DOMAIN job-names TO BE NAME OF JOB : STRING;
(NOTE: The domain job-names is assumed to be defined on a relation JOB which
contains all job names, but is not shown in Figure 15.)
CREATE DOMAIN person-names;
DEFINE DOMAIN person-names TO BE STRING;
CREATE RELATION EMP-NAME = EMP-NO DOMAIN employee-numbers
        NAME COMPOUND ( FIRST-NAME DOMAIN person-names,
        MIDDLE-NAME DOMAIN person-names NON-MANDATORY,
        LAST-NAME DOMAIN person-names) UNIQUE;
CREATE CONNECTION TYPE IDENTITY FROM EMPLOYEE (NUMBER)
        TO EMP-NAME (EMP-NO) MIN 1;
CREATE DOMAIN children-ages;
DEFINE DOMAIN children-ages TO BE INTEGER RANGE 1:25;
DEFINE DOMAIN sex TO BE {m, f};
CREATE RELATION CHILD = EMP-NO DOMAIN employee-numbers,
        NAME DOMAIN person-names AGE DOMAIN children-ages,
        SEX DOMAIN sex;
CREATE CONNECTION (child, parent) TYPE OWNERSHIP
        FROM EMPLOYEE (NUMBER) TO CHILD (EMP-NO);
CREATE DOMAIN department-numbers;
CREATE DOMAIN city-names;
CREATE RELATION DEPARTMENT = NUMBER DOMAIN department-numbers
        LOCATION DOMAIN city-names;
DEFINE DOMAIN department-numbers TO BE NUMBER OF DEPARTMENT :
        INTEGER;
DEFINE DOMAIN city-names TO BE NAME OF CITY : STRING;
(NOTE: The domain city-names is assumed to be defined on a relation CITY which
contains all city names, but is not shown in Figure 15.)
CREATE DOMAIN department-names;
CREATE RELATION DEP-NAME = DEP-NO DOMAIN department-numbers
        NAME DOMAIN department-names UNIQUE;
```

```
DEFINE DOMAIN department-names TO BE NAME OF DEP-NAME : STRING;
CREATE CONNECTION TYPE IDENTITY FROM DEPARTMENT (NUMBER)
        TO DEP-NAME (DEP-NO) MIN 1;
CREATE DOMAIN project-numbers;
CREATE DOMAIN security-codes;
DEFINE DOMAIN security-codes TO BE INTEGER RANGE 1:5;
CREATE RELATION PROJECT = NUMBER DOMAIN project-numbers
        SEC-CODE DOMAIN security-codes;
CREATE DOMAIN project-names;
CREATE RELATION PROJ-NAME = PROJ-NO DOMAIN project-numbers
        NAME DOMAIN project-names;
DEFINE DOMAIN project-names TO BE NAME OF PROJ-NAME : STRING;
DEFINE DOMAIN project-numbers TO BE NUMBER OF PROJECT : INTEGER;
CREATE CONNECTION TYPE IDENTITY FROM PROJECT (NUMBER)
        TO PROJ-NAME (PROJ-NO) MIN 1;
CREATE RELATION EMP-DEP = EMP-NO DOMAIN employee-numbers UNIQUE,
        DEP-NO DOMAIN department-numbers;
CREATE CONNECTION (department, employee) TYPE OWNERSHIP
        FROM EMPLOYEE (NUMBER) TO EMP-DEP (EMP-NO) MIN 1;
CREATE CONNECTION (department, employee) TYPE DIRECT REFERENCE
        FROM EMP-DEP (DEP-NO) TO DEPARTMENT (NUMBER) MIN 10;
CREATE RELATION DEP-PROJ = DEP-NO DOMAIN department-numbers,
        PROJ-NO DOMAIN project-numbers;
CREATE CONNECTION (project, department) TYPE OWNERSHIP
        FROM DEPARTMENT (NUMBER) TO DEP-PROJ (DEP-NO);
CREATE CONNECTION (department, project) TYPE OWNERSHIP
        FROM PROJECT (NUMBER) TO DEP-PROJ (PROJ-NO) MIN 1;
CREATE DOMAIN percents;
DEFINE DOMAIN percents TO BE INTEGER RANGE 0:100;
CREATE RELATION EMP-PROJ = EMP-NO DOMAIN employee-numbers,
        DEP-NO DOMAIN department-numbers, PROJ-NO DOMAIN project-numbers
        PERCENT-TIME DOMAIN percent;
CREATE CONNECTION TYPE OWNERSHIP FROM EMP-DEP (EMP-NO, DEP-NO)
        TO EMP-PROJ (EMP-NO, DEP-NO);
```

```
CREATE CONNECTION TYPE OWNERSHIP FROM DEP-PROJ (DEP-NO, PROJ-NO)
        TO EMP-PROJ (DEP-NO, PROJ-NO);
CREATE CONNECTION DERIVED (project, employee) TYPE OWNERSHIP
        FROM EMPLOYEE (NUMBER) TO EMP-PROJ (EMP-NO);
CREATE CONNECTION DERIVED (employee, project) TYPE OWNERSHIP
        FROM PROJECT (NUMBER) TO EMP-PROJ (PROJ-NO);
CREATE RELATION PROJECT-MANAGER = EMP-NO DOMAIN employee-numbers
        PROJECT-MANAGER DOMAIN project-numbers UNIQUE;
CREATE CONNECTION TYPE IDENTITY FROM PROJECT-MANAGER (EMP-NO)
        TO EMPLOYEE (NUMBER);
CREATE CONNECTION (project-managed, manager) TYPE DIRECT REFERENCE
        FROM PROJECT-MANAGER (PROJECT-MANAGED) TO PROJECT (NUMBER);
```

As we can see, the process of defining a model in the SMDL is tedious although straightforward. The advantage gained is that expression of queries is going to be simpler and more natural than, say, a relational calculus, or a relational algebra language such as SEQUEL,[Chea76]. Since the definition of a data model is usually done by experts, and is a one-time proposition, it makes sense to put considerable effort into it, and gain the rewards in simpler query language expressions. The query language (the SMQL) will be in constant use during the lifetime of operation of the database.

The choice of relation and connection names is a very important aspect of designing a structural model. These are the names that will be used by the SMQL. A careful choice of names will make the SMQL expressions look somewhat like natural language statements. We discuss the choice of names in Section 5.5.

The SMDL has been implemented in INTERLISP with the help of Rich Bagley. An interactive schema definition facility exists, which can be used to create a structural model schema. The SMDL system checks the structural model schema for errors and naming ambiguities.

In the next section, we present the SMQL.

4.3 THE STRUCTURAL MODEL QUERY LANGUAGE (SMQL)

4.3.1 Introduction

The SMQL is used for retrieval of information from, and update of information in, a database defined by a structural model. Retrieval and update are different operations. A retrieval operation specifies data to be retrieved from the database that may be related to other known data. If the operation is well defined, the retrieval request can be carried out and the desired data delivered from the database. If the operation is not well defined, either because the request does not make sense with respect to the represented situation, or because it *references* data that is not represented in the data model, no response can be given.

The general retrieval operation is not restricted to direct retrieval of data from the database, but can return to the user an indirect retrieval based on some function that is applied to the current database. One type of indirect retrieval returns a *Yes* or *No* (or *True–False*) value in response to a query that checks the database for the existence of some condition. Another type returns functions applied to numeric data.

An update operation can introduce new information into, remove outdated information from, or change existing information in the database. Update operations permanently modify the database, and hence should always be verified for correctness. That is why it is important to represent known constraints on the real-world situation in a data model. If an attempt is made to introduce erroneous data that violates these constraints, the error can be discovered automatically. Clearly, erroneous data can exist that *still satisfy* all the specified constraints, but recognition of this fact does not mean we should ignore solvable aspects of the problem.

Query languages for database systems exist that vary widely in terms of syntax, capabilities, mode of use, and intended type of user. This is unavoidable, since many database systems exist, which use diverse models, storage techniques, and access methods to the database.

We can categorise query languages according to several criteria:

(1) Procedurality: Some query languages are quite procedural in nature, so that a query is formed of many statements, and hence resembles a program. These languages are *procedural*, and are usually embedded in some programming language such as COBOL, FORTRAN, or PL/I. Other languages are *declarative* so that for most requests, a single query can be formulated to express the complete request.

(2) Mode of use: Highly procedural languages must be *embedded* in a *host* programming language. All access to the database is achieved by programs. Non-procedural languages can also be embedded in a programming language, but may also be used in a *stand-alone* mode, as a direct means of accessing the database. Hence, non-procedural languages can be used to access the database interactively from a terminal.

(3) Model: Most implemented systems are based on one of the hierarchical, network, and relational models. Hence, query languages can be categorised by the type of model they operate upon.

(4) Level of expected user: Languages exist at different levels. Some are meant to be used by programmers, others by knowledgable users who understand some aspects of a database computer system, and still others are meant to be used by naive users who know very little about computers.

Most current network and hierarchical languages are procedural, and involve a lot of details about the physical aspects of data storage and access. Hence, these languages are oriented *towards use by programmers and data processing professionals*. Programs can be written to efficiently perform frequent operations. Whenever new operations are required, new programs must by written to perform them. Ad hoc, and unforseen access to the database is difficult and costly.

Higher level languages for network systems have been proposed. For example, in [Brad78], a language that operates on network databases is described that is based upon the predicate calculus.

Relational languages have received the most interest because of their mathematical foundations. They are also at a higher level than the earlier hierarchical and network languages in the sense that there is less concern about physical access paths in the language. However, most implemented relational languages, such as SEQUEL [Chea76] and QUEL [WoYo76], still include statements that concern the physical storage of data.

Mathematically, relational languages are categorised into three broad classes [Ull79]: domain calculus languages, tuple calculus languages, and relational algebra languages. The syntax of both the domain calculus and the tuple calculus languages is quite close to the predicate calculus of mathematical logic [Kle67], and in that sense, both are highly non-procedural, or declarative languages. The difference between domain and tuple calculus is that in the domain calculus, attribute names and domain names in relations must be the same, and reference in the language is only to domain names. In tuple calculus, reference is made to tuples of relations (and hence the relations themselves), and domains are referred to within tuples. Other aspects of the two languages are identical. Both use variable names—domain variables in the domain calculus, and tuple variables in the tuple calculus—and both use the universal and existential quantifiers of the predicate calculus. The first relational calculus language ALPHA, was introduced by Codd [Codd71].

The relational algebra is quite different. Here, relations are operands, and algebraic operators are applied to them. Some operators are unary, such as *PROJECTION* (select attributes from a relation to form a new relation) and *SELECTION* (select tuples from a relation that satisfy a certain condition). Other operations are binary, such as *JOIN* (combine two relations on a certain attribute) and *CARTESIAN PRODUCT*. Set operators such as *UNION* and *INTERSECTION* are applicable when relations are of the same *a-rity* (same number of attributes).

The three relational language classes have been shown to have the same expressive power (see [Ull79] for a proof). A relational language which possesses at least the same power is termed *relationally complete* by Codd [Codd72a].

Implemented relational languages are mostly of the relational algebra class, although the language constructs vary widely.

Relational calculus languages are geared for mathematically-oriented users, with a good background in mathematical logic. A naive user with limited mathematical backgroung would find it difficult to express a query because of the extensive use of variables in the language, and the way quantifiers are nested. However, the formal nature and power of expression of the relational calculus makes it quite attractive. This has led to attempts to develop interactive languages based on the relational calculus which guide the user through query expression, such as QBE [Zlo75].

Relational algebra languages are also geared to mathematically-oriented users. The relational algebra operations PROJECTION and SELECTION have a direct correspondence to real-world structures. The two operators can correspond to selecting a set of properties of an object, and selecting a set of objects from an object class. However, the JOIN operator, which relates tuples from different relations, is more difficult to conceptualise by naive users [Reis77]. The users must also have complete knowledge of the relational schema data structures, and the meaning of each relation when preparing their queries.

Research into high-level languages produced methods which free the user of the burden of having to know the detailed data structures of a relational schema ([CaKa76], [Sag77], [ChaKe78]). In these approaches, users do not have to know the relation where an attribute resides, and hence the users do not have to explicitly specify joins. The problem with this approach is that in some cases, ambiguity is encountered when several methods exist for joining relations.

Several systems have been implemented which provide the user with a limited natural language interface (LIFER [Heea78], ROBOT [Har77], Rendevous [Coea78], PLANES [Wal78]). These systems include some form of catalog which includes information on the domain of discourse. Attempts are being made to use the information in the database itself rather than maintaining a separate catalog for the natural language interface [DaKa80].

Requests may be ambiguous when expressed in a natural language, so some techniques for detecting ambiguity must exist. For example, the Rendevous system interacts with the user to disambiguate requests. Natural language systems are still in the experimental stage, and current systems are quite inefficient.

More recently, a class of *functional* languages, which can operate on a variety of models (augmented with some additional information) has been proposed. The languages are claimed to be more natural to use. The basic model this class of languages

operates on is a *functional* or *binary relationship* model, where all relationships are between pairs of domains. An example is the DAPLEX language [Ship79], which is object, or entity, oriented. Another functional language, FQL [BuFr79] operates on a network model.

Our language is also *object-oriented*. The mechanism by which we select a relation that is connected to another relation is quite similar to the functional language mechanism. However, our tuple selection mechanism is quite different from DAPLEX. The tuple selection mechanism is similar to the *SELECTION* operation of a relational algebra, but more general since restriction predicates can be applied to attributes from connected relations, as well as to the attributes of the relation under considerations.

We can also consider the attribute selection mechanism as performing a *SELECTION* operation on a relation that is a *JOIN* of several connected relations. However, the *JOIN* operations are implicitly specified by the connection names.

4.3.2 Components of a query

In this section, we discuss our ideas about what the components of a real-world oriented query should be. Hence, we refer to a model of the world that is composed of object classes, properties of object classes, and relationships between object classes, similar to our presentation in Chapter 2. Then we present the main components of a query in the SMQL, and discuss its correspondence to our concept of real-world queries.

In our model of the real-world, objects are grouped into object classes according to their structure. We will assume that each single query will be interested in retrieving similar information about a number of objects from a single object class. Hence, the query must identify three main components:

(1) The object class under consideration.

(2) The subset of objects from the class that are of interest.

(3) The properties whose values are desired for each specified object.

The object class under consideration is specified by its name. A query will specify an object class O as the object class under consideration.

To select a subset of objects from the class, we specify a condition that is applied to each object $o_i \in O$. The subset of objects $O_S \subseteq O$ that satisfy the condition will be the ones selected.

Finally, the properties are specified by their names. In general, we may want to retrieve values of properties of the object class itself, or values of properties from other object classes that are related to it. Hence, we specify a set of properties P such that each property $p_i \in P$ is either:

(a) A property of the specified object class O, or

(b) A property of another object class O_R that is related to object class O by some relationship. In this case, we specify the object class O_R, and the relationship R that relates it to O.

The query will return for each selected object o_i the values of each specified property for that object. For a property p_i from an object class O_R, the objects in O_R that are related to o_i by the relationship R are determined, and the values for p_i are retrieved from these tuples. Hence, the value returned for a single specified property for an object o_i may be a single value or a set of values, depending on the characteristics of the property.

(a) If the property is of the object class O, a single value is returned if the property is not repeating, and a set of values is returned if the property is repeating.

(b) If the property is of the object class O_R, a single value is returned if the property is not repeating and the cardinality of the relationship R relates a single object from O_R to an object of O. A set of values is returned if the property is repeating, or if the cardinality of the relationship R relates a set of objects from O_R to an object of O. In the latter case, the set of values returned is the union of the value of the property (or values if repeating) for all objects in O_R that are related to the (single) selected object of O.

To allow more generality, a single query may want to retrieve information from objects in different object classes that are related together in some general way. In this case, two or more object class names are specified. The query will take an object from each class and see if the conditions for object selection are met. If so, a single combined object is returned that includes the requested values of the properties from the objects (one from each class). Hence, all combinations of single objects, one from each class (the cross product of the object classes), are considered individually.

Formally, we examine each combination $o1_i, o2_j, \ldots$ of objects from the object classes $O1, O2, \ldots$, for $i = 1, \ldots n(O1)$ and $j = 1, \ldots n(O2)$, where $n(Ok)$ is the number of objects in class Ok . For each combination that satisfies the conditions, we return a single object with the desired properties from $o1_i, o2_j, \ldots$.

When a set of values is specified to be returned for a property, we may not want all the values returned, but only a restricted set of the values. Hence, the general specification of a property whose values are to be retrieved should allow the specification of a condition that selects a subset from the set of values.

A condition for selection of the set of objects O_S from O is applied to values of properties of the objects by comparing them to known, given values. Each object whose current values for properties satisfy the condition are selected for retrieval. The known values may be the result of retrieval of values from properties of other objects—the result of another query.

We will specify the form of conditions precisely in Section 4.3.3.

Now we consider the form of a query in the SMQL. The names of relations, attributes, and connections are used by the SMQL to specify a query. In general, there will correspond to each object class from the real-world a single relation whose tuples

correspond directly to the objects of the class. We call this relation the *main relation* that represents the object class (see Section 5.2). Hence, giving the name of this relation specifies the object class under consideration.

Connections of the structural model are used to link tuples from one relation to tuples in another relation. To specify the tuple(s) in a relation R_1 that are connected to a tuple in another relation R_2, we use the connection name in the direction from R_2 to R_1. Recall that all our connections are directed arcs, and a connection can have two names. The first name corresponds to the given direction of the connection, and the second name to the reverse direction (Section 3.2.2).

Properties are represented by attributes in the structural model. Hence, to refer to the properties whose values are to be retrieved, we specify the appropriate attribute names. When attributes to be retrieved are not in the main relation of the object class under consideration, we use the connection names to specify the path from the main relation to the relation in which the attribute is located.

Although the structural model formalisms do not correspond directly to our perception of the real-world structures, as presented in Chapter 2, the SMQL is designed to closely correspond to such a perception. By a careful choice of the names of attributes, relations, and connections (see Section 5.5), the statements of the SMQL will closely resemble our perceptions. We refer to the main relation that represents an object class by its name, which should be the same as the name of the object clas.. Similarly, we can refer to objects from a class that are related to an object of another class by specifying the name of the connection that represents the relationship. This name should match the name the user associates with that relationship.

There are three types of SMQL statements: <retrieve statement> (or query), <update statement>, and <define statement> . So far, we have briefly discussed the <retrieve statement> only, which we also call an SMQL *query*. The <update statement> is used to update the database, and the <define statement> is used to augment a data model with additional structures. We discuss these two statements in Section 4.3.4 and Section 4.3.5 respectively.

The syntax for a statement in the SMQL is the following:

<SMQL statement> ::= <retrieve statement> | <update statement> | <define statement>
<retrieve statement> ::= <GET clause> <WHERE clause>

An SMQL query (or <retrieve statement>) is formed from two clauses: the GET clause and the WHERE clause. The GET clause specifies the main relation(s) we are considering (the object class(es) under consideration), and the data attributes whose values are to be retrieved from the selected tuples in these main relations (the properties of the selected objects whose values are desired). The WHERE clause selects the subset of tuples from the main relation(s) from which the data values are to be retrieved (the subsets of objects from the object classes that are of interest) by specifying the conditions to be satisfied by a selected tuple.

The attributes specified in the GET clause are either from the main relation(s), or are from other relations that are connected to the main relation(s). In the latter case, we specify one or more connection names to give the desired path between the two relations.

4.3.3 The GET and WHERE clauses

We now give the syntax and semantics of the two main clauses in the SMQL, the GET clause and the WHERE clause.

4.3.3.1 *Syntax of the* GET *and* WHERE *clauses*

First consider the GET clause.

```
<GET clause> ::= GET <retrieve spec> {, <retrieve spec>}
       [ORDER BY <attribute spec list>]
<retrieve spec> ::= <attribute spec list> of <relation name>
```

The GET clause specifies one or more <relation name>'s that will in general specify one or more main relations which represent object classes. For each of these relations, the properties whose values will be returned for the selected tuples (tuples are selected in the WHERE clause) are specified in the <attribute spec list> .

The ORDER BY clause is optional, and can be specified only on attributes whose domains are ordered. This clause only serves to order the tuples in the output based on the values of the specified attributes.

Next, we give the syntax for the attribute specifications.

```
<attribute spec list> ::= <attribute spec> | '(' <attribute spec> {, <attribute spec> }')'
<attribute spec> ::= ( <single spec> | <attribute spec list> ) { of <connection name>}
      [ :<logical expression> ]
<single spec> ::= [ <function> | <attribute name> | EXISTS | COUNT
<function> ::= EXISTS | COUNT | TOTAL | AVERAGE | SD |
      ( MAX | MIN ) '(' <expression> ')'
```

An <attribute spec list> specifies a list of attributes. Some of the attributes are of the relation specified in <relation name> of the GET clause, while other attributes are from relations connected to the relation specified in <relation name> by one or more connections.

An attribute from the relation specified in <relation name> is designated by the attribute name only. An attribute from a relation that is connected to the relation specified in <relation name> is designated by the attribute name, plus a list of connection names that define a path from the relation specified in <relation name> to the relation from which the attribute is.

The <logical expression> is used to select a subset of the values of an attribute that will return a set of values, if needed. The syntax for <logical expression> and <expression> are given below.

The functions TOTAL, AVERAGE, and SD (standard deviation) can be applied only to numeric attributes. The functions MAX and MIN can be applied only to an attribute whose domain is ordered.

The function COUNT is applied to a set, and returns the number of elements in the specified set. It may be specified on connection or relation names only, to return the number of tuples in a relation.

The function EXISTS is also applied to a set, and returns TRUE or Yes if at least one element exists in the specified set. It may also be specified on connection or relation names only, and returns TRUE or Yes if at least one tuple exists in a relation.

We will discuss these functions in Section 4.3.6. These functions are standard in many query languages, and are used to return values that are transformations of values obtained from the database.

Now consider the WHERE clause. The WHERE clause is used to specify a logical expression, which is formed from a number of conditions. This logical expression is applied to a combination of single tuples, one tuple from each of the relations specified in the GET clause. The logical expression will include references to attributes of the relations of the GET clause, and to attributes of relations connected to the relations of the GET clause. If the combination of tuples satisfies the logical expression—meaning that the logical expression evaluates to TRUE when the values of attributes of the tuples in the combination are substituted in the logical expression—the combination of tuples is *selected*. This procedure is repeated for all combinations of tuples.

The syntax of the WHERE clause is specified as follows.

```
<WHERE clause> ::= WHERE <logical expression>
<logical expression> ::= [ <logical expression> ( AND | OR ) ] <logical term>
<logical term> ::= [ NOT ] ( '(' <logical expression> ')' | <condition> )
<condition> ::= <set expr> <set comparison op> <set expr> |
      <single value expr> <value-set comparison op> <set expr> |
      <single value expr> <single value comparison op> <single value expr>
<set comparison op> ::= ⊆ | ⊇ | = | ≠
<value-set comparison op> ::= ∈ | ∉
<single value comparison op> ::= = | ≠ | <numeric-ordered op> | <alphanumeric-date op>
<numeric-ordered op> ::= < | ≤ | > | ≥
<alphanumeric-date op> ::= BEFORE | AFTER
<single value expr> ::= <expression>
<set expr> ::= <expression>
```

The WHERE clause is formed from a logical expression, which is formed from conditions grouped using the usual logical connectives AND , OR , and NOT. Evaluation is *left to right*, with parentheses used for grouping. A condition returns TRUE or FALSE for each combination of tuples that is formed from one tuple from each of the relations specified in the GET clause. The logical expression is then evaluated for each combination of tuples, and a TRUE value means that the combination is selected.

A condition compares two expressions. An expression returns either a single value, or a set of values. Different comparison operators apply to the possible combinations of single valued expressions and set valued expressions in a condition. Hence, there are three types of comparison operators.

The <single value comparison op> compares two single values in a condition. Additional operators apply if the domain is numeric, ordered, date, or alphanumeric. The <value-set comparison op> compares a single value with a set of values, so these operators are not symmetric with respect to the two operands. The <set comparison op> compares two sets of values. We also allow a single value to be specified for set comparisons, in which case the value is transformed to the set that contains that value. We will further discuss the semantics of the WHERE clause in Section 4.3.3.2.

The syntax for expressions follows. Arithmetic can be applied to numeric expressions, whether sets or single values. If applied to a set, a single value must be the other operand, and the operation—using that operand—is applied to all members of the set, creating a new set of values as the result.

In the following syntax, a <single value> can be derived only from a <single value expr>, and a <value set> from a <set expr>. We could specify this precisely in the syntax at the expense of some repetition, but chose not to.

```
<expression> ::= [ <expression> ( + | - ) ] <term>
<term> ::= [ <term> ( × | ÷ |DIV|REM ) ] <factor>
<factor> ::= '(' <expression> ')' | <value>
<value> ::= <single value> | <value set>
<single value> ::= <database value> | <constant> | <val-op> <value set>
<val-op> ::= EXISTS | COUNT | TOTAL | AVERAGE | SD |
        ( MIN | MAX ) [ '(' <single value expr> ')' ]
<value set> ::= [ <value set> <set operator> ] <basic value set>
<basic value set> ::= '(' <value set> ')' | <database value> |
        <constant> | '{' <constant> {, <constant>}'}'
<set operator> ::= ∪|∩|−
<database value> ::= <attribute spec list> [ of <relation name> ] |
        '(' <retrieve statement> ')'
<constant> ::= any constant number or string
```

Values are either specified explicitly by constants or sets of constants, or are specified as the value of some attribute or retrieval operation from the database—a <database value>. In the latter case, we cannot tell from the language syntax whether the value to be retrieved will be a *single value* or a *value set*. However, we can tell by looking only at the schema whether to expect a single value or a set of values.

A <database value> is either specified to be the value of some attribute from, or connected to, the relations of the GET clause, or a complete retrieval request that is independent from the relations specified in the GET clause. In the former case, if only a single relation name appears in the GET clause, then the relation name in the WHERE statement is optional since it must match the single relation name in the GET clause.

Sets can be *formed from other sets using* a <*set operator*>. In this case, we assume that both sets are from the same domain of values. Single values can be formed from sets by applying a <value-op> to the set.

We now discuss the semantics of a query in the SMQL by presenting the semantics of the GET clause and the WHERE clause.

4.3.3.2 Semantics of the GET and WHERE clauses

The GET clause is of the form:

Q1: GET $A_{11}, \ldots A_{1i}$ of R_1, $A_{21}, \ldots A_{2j}$ of $R_2, \ldots$ $A_{n1}, \ldots A_{nk}$ of R_n

where we have *n* relations, and for each relation a number of attributes are specified.

An attribute A_{lm} is specified on relation R_l. *It is either an attribute of* R_l, or an attribute of some relation R_e that is connected to R_l . In the former case, only the name of the attribute A is specified. The *value* of A_{lm} for a *given tuple* in R_l is the current value of the attribute A for that tuple.

In the latter case, the name of an attribute from some relation R_e is specified, followed by one or more connection names, so the form of A_{lm} is

A of cname_t of ... of cname_1 .

In this case, a connection path is traced from R_l via cname_1 to ... to cname_t to R_e . We will not specify the algorithm for this here, but leave it for Section 4.3.7. However, from the algorithm, we will know whether a tuple from R_l is connected to a single tuple or to a set of tuples in R_e , by tracing the cardinality of the connections. The *value* of A_{lm} for a given tuple in R_l is the set of current values of A for the tuple(s) in R_e that are connected to the given tuple in R_l .

If it is determined that a set of tuples from R_e is connected to a single tuple in R_l, we *may specify* a *further* condition to select only a subset of these tuples. In this case, the *value* of A_{lm} for a given tuple in R_l is the set of current values of A for the tuple(s) in R_e that are connected to the given tuple in R_l and that *satisfy* the given condition. (See below *for the meaning of satisfy.*)

For a given query, a number of tuples from each R_l in the GET clause will be *selected* for retrieval by the WHERE clause (see below), and for each of these tuples the values of each A_{lm} is returned as discussed above.

Now consider the form of the WHERE clause. The WHERE clause is formed from a logical expression, which is formed from a number of conditions. Each condition should include a reference to one or more of the relations $R_1, \ldots R_n$ of the GET clause. This reference is specified in the syntax by the production rule:

<database value> ::= <attribute spec list> | of <relation name>] .

The reference will be to some attribute A_{lm} specified on relation R_l. The reference is either to an attribute of the relation R_l, or an attribute of a relation R_t connected to R_l via a connection path as discussed above. The reference could also be to some function applied to the specified attribute.

The condition compares this reference to attribute A_{lm} to some other value, or set of values, which could be one of the following:

(1) An explicit constant value, or set of constant values.

(2) A value or set of values that is specified by a complete retrieval from the database.

(3) A value or set of values that is specified by some other reference to a relation R_k in the GET clause.

(4) A function applied to any of the above cases.

Now, the semantics of a complete query are as follows:

For each combination of tuples that consists of one tuple from each of the relations of the GET clause, or formally for each element of the set

$$\{t_1, \ldots t_n | t_1 \in R_1, \ldots t_n \in R_n\}$$

each condition is evaluated to either true or false. The complete logical expression of the WHERE clause, which is composed of conditions connected by the logical connectives AND and OR , is then evaluated to either true or false. If it evaluates to true for a particular combination of tuples, a single, unnormalised tuple t_r, formed from the values of the attributes of the GET clause for the tuples $t_1, \ldots t_n$ is returned. The output of the query is the unnormalised relation consisting of the set of all such tuples t_r.

Consider the semantics of a single condition. Recall that a condition is applied once to each combination of tuples from the GET clause, and can evaluate to true or false for each combination. A condition can compare two single values, two sets of values, or a single value and a set of values. The comparison operators are different for each case.

There are three types of comparison operators. The first type of operator compares two sets of values together, and includes the standard set comparison operators: $\subseteq, \supseteq, =, \neq$. The second type compares a single value with a set of values, and the appropriate operators here are $\in$ and $\notin$, which test whether the single value is in the set of values or not, respectively. Note that these two operators are not symmetric with respect to the two operands. The third type of operator compares two single values. We assume the two values are of the same domain, or of comparable domains. The operators now are dependent on the domain of the two values. However, two operators

are domain independent, which test for equality of the two values. These operators are $=, \neq$.

A condition is *well-defined* if the two operands in the condition are comparable by the comparison operator of the condition. A well-defined condition evaluates to true if the two operands satisfy the specified comparison operator.

A logical expression evaluates to true if the expression, with each condition evaluated to true or false, evaluates to true according to the well known rules for combining truth values using the logical connectives AND and OR .

A set can be formed from other sets using the standard set operators that form the intersection, union, and difference of two sets: $\bigcap, \bigcup$ and $-$. A single value can be formed by applying the unary operator COUNT to a set, which returns the number of elements in that set. This value can then be used as a single value in a comparison.

Other functions return single values when applied to sets or multisets of elements that are of a specific domain. We discuss these functions in Section 4.3.6.

We also have single-value comparison operators that depend upon the low-level domain of the values being compared, and hence are domain specific. For a numeric domain, or in general for any domain upon which some total order is defined, the single value comparison operators are expanded to include the standard comparison operators: $<, >, \leq$, and $\geq$.

The single values can be of several low-level domain types: numeric (integer or real), date, alphabetic or alphanumeric character strings. The alphabetic and alphanumeric orderings upon which the single-value comparison operators are specified are the lexicographic orderings. For alphabetic or alphanumeric string ordering, we use the operators BEFORE for $<$, and AFTER for $>$ in the lexicographic ordering, and similarly for the domain date.

The absence of a WHERE clause in a query is interpreted to mean that the clause is WHERE true, which satisfies all combinations of tuples in the relations of the GET clause. If, on the other hand, the logical expression evaluates to false for all combinations of tuples, then the output of the query is empty.

We have now discussed the GET and WHERE clauses, which are the components of the <retrieve> statement of the SMQL. We will present the mechanism for tracing connections in detail in Section 4.3.7. Next, we present the two remaining statements of the SMQL. In Section 4.3.4, we present the <update statement>, which changes the content of the database, and in Section 4.3.5, we present the <define statement>, which is used to define derived structures, update transactions, and domain constraints.

4.3.4 The <update statement>

The <update statement> provides us with the capability to change the contents of the database. Whenever an update operation is specified, an automatic check should be generated by the database management system to insure that the update operation does not violate the structural constraints specified in the data model schema.

There are three basic update operations: changing the value of an attribute in a tuple, deleting a tuple from a relation, and inserting a tuple into a relation.

The CHANGE operation requires us to specify four things: a relation, the tuple or set of tuples in that relation which will be affected, the attribute whose value is to be changed, and the new value for this attribute. The DELETE operation requires us to specify two things: a relation, and the tuple or set of tuples that will be deleted from this relation. Hence, we include a WHERE clause for both of these operations to specify the set of tuples to be affected. These tuples are defined to be the tuples that satisfy the condition of the WHERE clause.

The INSERT operation only requires us to specify a relation name, and the new tuple that will be inserted in the relation. The syntax for these three operations follows.

```
<update statement> ::= ( <CHANGE clause> | <DELETE clause> ) <WHERE clause> |
        <INSERT clause>
<CHANGE clause> ::= CHANGE <attribute spec> of <relation name> TO <value set>
<DELETE clause> ::= DELETE FROM <relation name>
<INSERT clause> ::= INSERT <relation name>: ( <value tuple> |
        <attribute name> <single value> {, <attribute name> <single value> } )
<value tuple> ::= '<' <single value> {, <single value> } '>'
```

In the change operation, a single attribute that is either of the relation specified in <relation name>, or of a relation connected to the relation specified in <relation name>, is designated in the CHANGE clause. This attribute is changed to the value specified in <value set> for all the tuples that satisfy the WHERE clause.

To delete a tuple, we need only specify the relation name in the DELETE clause. All tuples in the relation that satisfy the WHERE clause are deleted.

Note that both update and deletion are subject to the constraints specified in the data model. If the change or deletion will cause the database to violate some constraints, it is aborted, and the user notified. Furthermore, deletion of a single tuple may trigger the deletion of tuples in other relations that are owned by that tuple. The algorithms for maintaining structural consistency have been given in Section 3.3.2.

Insertion is straightforward, since all we have to specify is the relation name where the tuple will be inserted, and the new tuple itself. Insertion is also subject to the structural constraints, and any attempt to insert a tuple that makes the database inconsistent is aborted.

In some cases, we may have to insert several related tuples in one *transaction*. The transaction is defined by the <define statement>, discussed in the following section. For an *insertion transaction*, the check that the consistency constraints of the data model are not violated is made after the complete transaction. A transaction is allowed if the database is consistent upon terminating all the insertion operations even though temporary integrity violations may have occurred. Update transactions can also include deletion and update operations.

In the next section, we discuss the <define statement>.

4.3.5 The <define statement>

The <define statement> is used to define three things: update transactions, derived structures, and domain (or value) constraints. The <define statement> is part of both the Structural Model Definition Language (SMDL, see Section 4.2) and the SMQL.

```
<define statement> ::= <define update transaction statement> |
        <define derivation statement> | <define constraint statement>
```

An update transaction defines a sequence of basic update operations that are frequently executed as a unit. Checking for integrity violations is carried out at the end of the completed transaction. The syntax for the <define update transaction statement> is the following:

```
<define update transaction statement> ::= DEFINE TRANSACTION <name>
        '(' <parameters> ')' : <transaction> {; <transaction> } END
<parameters> ::= <parameter> {, <parameter> }
<parameter> ::= '(' <parameters> ')' | <name>
<transaction> ::= <update statement>
<name> ::= any character string
```

A transaction is defined by its name, parameter list, and sequence of insert, delete or update operations. When invoked, the list of values in the parameter list for that invokation replace the parameter names in the transaction definition. This is similar to call by value parameters for procedures in programming languages. The transaction is aborted if it would result in an inconsistent database.

Now we consider the statement used to define a derived structure. A derived structure is a relation, connection, or attribute that is defined in terms of other structures of the data model. Hence, defining a derived structure is a means of easy reference to a frequent retrieval request. We now give the syntax for the <define derivation statement>.

```
<define derivation statement>  ::= <relation derivation> | <connection derivation> |
        <attribute derivation>
<relation derivation>  ::= DEFINE DERIVED RELATION <relation name> TO BE
        <retrieve statement>
<connection derivation>  ::= DEFINE DERIVED CONNECTION <connection definition>
<attribute derivation>  ::= DEFINE DERIVED ATTRIBUTE <attribute name>
        of <relation name> TO BE ( <attribute spec> [ of <relation name> ]|
        <retrieve statement> )
```

A derived relation is simply a retrieval statement given a name. Hence, it can be an unnormalised relation. A derived connection is specified by a connection definition, as in the SMDL. A check is made to ensure that the type of connection corresponds to the combination of connections of some path that exists in the data model. A derived attribute is specified by an attribute specification that is connected to the relation, and may also be repeating. The optional <relation name> in the derived attribute specification must be the same as the other <relation name> if used. A derived attribute may also be specified by a general retrieval statement.

Finally, consider the specification of domain constraints.

```
<define constraint statement>  ::= DEFINE CONSTRAINT on <relation name>:
        <logical expression>
```

A domain constraint is specified by a logical expression on a relation. Every tuple in the relation must satisfy that constraint. Such constraints should be specified only if they are absolutely necessary to enforce, since they can be quite expensive to check. These constraints are similar to integrity assertions [Ston 74].

4.3.6 Functions and arithmetic

There are several built-in functions in the SMQL which apply to attributes of numeric domain. Similar functions exist in several query languages, such as SEQUEL [Chae76] and DAPLEX [Ship79]. The functions are TOTAL, AVERAGE, SD (standard deviation), MAX, and MIN. The aggregate numeric functions TOTAL, AVERAGE, and SD always apply to multisets, so duplicate values are taken into account. This is so because that is the normal usage for such functions. If we want to apply them to sets without duplicates, we can always create a derived relation that retrieves the set of unique values, and then apply the function to that relation (see Section 4.3.9 for examples).

Two other standard functions—MAX and MIN—not only apply to numeric domains, but also to any ordered domain. The functions MAX and MIN return the maximum and minimum values from a multi-set of values. For attributes with ordered domains, MAX returns the highest value according to the order from the multi-set of values that it is applied to, and MIN returns the lowest value from that multi-set. We can also

use these functions to return the i^{th} largest or i^{th} smallest element in a multi-set, by placing an (i) following the MAX or MIN operator.

We also allow arithmetic on numeric valued attributes. For all numeric attributes, the arithmetic operators are $+, -, \times, \div$. For integers, we also include DIV and REM, which return the quotient and remainder of an integer division operation, respectively. When an arithmetic operation is applied on a single value and a set, it returns the set of values that results from applying the operation using the single value to each value in the set. Arithmetic operators are standard in many query languages.

There are two more standard functions, COUNT and EXISTS. The COUNT function is applied to a set of values of any domain, and returns the number of values in this set. The COUNT function may also be applied on a connection name only, in which case it returns the number of tuples connected to a specified set of tuples via that connection. The COUNT function can also be applied over a relation name, in which case it returns the number of tuples in that relation. Examples are given in Section 4.3.9.

The EXISTS function is also applied to a set, and returns TRUE or Yes if at least one element is in the set. If the EXISTS function is applied to a set in the GET clause—which could be an attribute that returns a set of values, or a connection which is connected to a set of tuples, or a relation which includes a set of tuples—it returns Yes or No to the output of the query. If the EXISTS function is applied to a set of values in the WHERE clause, it evaluates to TRUE or FALSE in the logical expression of the WHERE clause.

4.3.7 Mechanism for tracing connections, and non-ambiguity of the SMQL

In this section, we describe the procedure that is taken to trace an attribute that is not of the relation under consideration, but of a relation connected to the relation under consideration via a connection path. We then give rules that should be imposed by the schema processor of the database management system to ensure that the names chosen for attributes, relations, and connections do not permit ambiguity in the specification of a SMQL statement.

It is important that a query language be unambiguous, especially for a formal database language. Non-ambiguity solves the problem of multiple paths that exists in some high-level languages which permit the specification of attributes without necessarily specifying from which relation they are [Sag77, CaKa76]. The multiple-path problem exists because the data model used in these systems is a pure relational model, which does not contain any connection information. A pure relational schema contains less information than a structural schema. In a relational schema, numerous paths which connect, or join, two relations may exist. A heuristic approach to select the probable intended path is applied in these systems, which may work quite well in general but does not guarantee the correct retrieval. This problem of multiple paths also exists in network schemas.

In our approach, the user is forced to specify the path, although the path specification is in very natural terms. If a path is frequently used, it can be predefined by a derived structure, so that it does not have to be explicitly specified in future references.

Before we give the rules for non-ambiguity in path specification, we describe the mechanism for tracing a sequence of connections. The non-ambiguity rules are defined particularly to make the tracing of a sequence of connection names from a given relation to an attribute name in some other relation unambiguous.

4.3.7.1 *The mechanism for traversing connections*

Recall from Section 4.3.3.2 that in both the GET and the WHERE clauses, we encounter attributes A_{lm} that are specified on some relation R_l whose name appears in the GET clause. The attribute A_{lm} is either an attribute of R_l, or an attribute of some relation R_e which is connected to R_l via a connection path from R_l to R_e. In the latter case, the name of the attribute from relation R_e is specified, followed by one or more connection names, so the form of A_{lm} is

A of cname_t of ... of cname_1 .

The connection path is traversed from R_l via cname_1 to ... to cname_t to R_e . We now specify the algorithm for traversing such a connection path to locate R_e. First, we give some definitions.

Definition 15 is just a reiteration of the rule for connection names. Definition 16 defines a connection name path, which is a sequence of connection names. This is different from a connection path, which is a sequence of connections. A single connection name in a connection name path may in general specify more than one connection (with the same name) in a connection path.

Definition 15: (Names of connections) Consider a connection whose direction is from relation R_1 to relation R_2, and whose name is (cname_1,cname_2). Then, the connection name *from* relation R_1 is cname_1, and the connection name *from* relation R_2 is cname_2. Similarly, the connection name *to* relation R_1 is cname_2, and the connection name *to* relation R_2 is cname_1.

Definition 16: (Connection name paths) Consider an attribute specification A_{lm} specified on the relation R_l which is an attribute A of a relation R_e connected to R_l via a connection path from R_l to R_e. Hence, A_{lm} of R_l has the form

A of cname_t of ... of cname_1 of R_l .

Then, $\text{cname}_1 \ldots \text{cname}_n$ is called a *connection name path* from R_l to R_e, and cname_n is called the *last connection name* on the connection name path.

Now, we define the inheritance rules for subrelations. Attributes and connections from a base relation are automatically referenced from a subrelation. We give the rules by which such a reference is carried out. This involves the enforcement of some rules on the names of attributes and connections in the schema to assure non-ambiguity. We give the non-ambiguity rules in the next section.

Definition 17: (Superrelations) A relation R^S is a *superrelation* of relation R at *level* i if a sequence of i identity connections exist from R to R^S such that all the connections are in the direction from R to R^S. A relation R^S is a *superrelation* of relation R if some i exists to satisfy the above condition.

Hence, a superrelation of relation R at level 1 is a base relation of R, and at level 2 is a base relation of a base relation of R, and so on. Put in another way, a superrelation of R is a relation such that R is a subrelation of it at some level.

Definition 18: (Inheritance of attributes for subrelations) An attribute name A is *directly reachable* from relation R_l if:

(a) Some attribute of R_l has name A, or

(b) An attribute of some superrelation R_b of R_l at level n has name A, and no attribute in R_l or in any superrelation $R_{b'}$ of R_l at level i, $0 < i < n$, has the same name.

A similar definition exists for connections.

Definition 19: (Inheritance of connections) A connection name cname is *directly reachable* from relation R_l if:

(a) Some connection c has name cname in the direction from R_l, or

(b) A connection c of some superrelation R_b of R_l at level n has name cname in the direction from R_b, and no connection has name cname from R_l or from any superrelation $R_{b'}$ of R_l at level i, $0 < i < n$.

Next, we give the algorithm. Informally, the input to the algorithm is an attribute name, followed by a connection name path, followed by a relation name. What the algorithm does is that for each connection name in the connection name path, starting from the given relation, it traverses one or more connections that have that name in the appropriate direction until it reaches a relation that does not have a connection leading from it with that name. Then, the next connection name in the connection name list is taken, and starting from the last relation reached, the procedure is repeated.

Hence, each connection name (except the last one) traverses a path of at least one connection. Each connection name also traverses a sequence of relations which ends at some relation that becomes the starting relation for the next connection name. For each sequence of relations traversed using a single connection name, we must check for the possibility of a loop. If we encounter the same relation twice in the sequence, we end the sequence the second time we encounter the relation (otherwise we get into an infinite loop).

For the final connection name, we search each relation in the sequence in the order the relations are found for an attribute with the desired name which is directly reachable from the relation.

When we search if an attribute name A is directly reachable from some relation R , we first examine the attributes of R itself. If we do not find the desired attribute name, we must look at all relations that R is a subrelation of for the desired attribute. This is due to the nature of subrelations, since they represent subsets in most cases, and objects in the subset inherit all properties of the main object in the base relation. We must do the same when searching for a connection name leading from a relation.

The connection traversal algorithm:

Input:

An attribute specification of the form

A of $cname_t$ of ... of $cname_1$ of R_i .

Output:

A triple (R_e,con-list , x), where

R_e	is the relation from which A is directly reachable,
con-list	is the connection path from R_i to R_e,
x	is 1 *if* a single tuple in R_e is connected to a tuple *of* R_i
	is N if a set of tuples in R_e is connected to a tuple in R_i.

or, failure if the algorithm could not locate the relation R_e from which attribute A is directly reachable.

The algorithm:

Variables used:

t	t is the number of connection names in the connection name path
con-list	con-list will hold the connection path from R_i to R_e
R	R will traverse relations in the path from R_i to R_e
rel-list	rel-list is used to check for connection loops
curcname	curcname will hold the current connection name under consideration
i	i will advance connection names

Functions and procedures used (defined after the algorithm steps):

(1) reach (A, R)

Takes an attribute name A and a relation name R, and returns TRUE if some attribute with name A is directly reachable from relation R, and false otherwise.

(2) f (cpath , R_1, R_2)

Takes a connection path cpath from relation R_1 to relation R_2, and returns 1 if a tuple from R_i ia connected to a single tuple in R_2, and N if a tuple from R_1 is connected to a set of tuples in R_2.

(3) reachc (curcname , found , R, c)

Takes a connection name curcname and a relation name R, and returns TRUE in found if curcname is directly reachable from R (in this case, it also returns the found connection c and in R returns the relation to which c is connected); otherwise returns FALSE in found .

Algorithm steps (square brackets ([]) are used for grouping):

(0) (Initialize) Set con-list to empty, R to R_i, rel-list to R , curcname to undefined, i to 1, t to the number of connection names.

(1) If $t = 0$ then (if reach (A, R) then return (R_i, con-list , 1) else return failure)

(2) Set curcname to $cname_i$, increase i by 1. *If* $t = 1$, go to step 4.

(3) ($t > 1$; find a connection path for some $cname_i$ (a connection name that is not the last connection name on the connection name path).)

Call reachc (curcname , found , R , c).

(3.1) (Traverse one connection for $cname_i$.) If found add the connection c to con-list , and *traverse the connection to relation* R' *at the other end of the* connection.

If R' is in rel-list (this means we have encountered a connection loop), decrease t by 1, set rel-list to R', set R to R', and go to step 2;

otherwise add R' to rel-list , set R to R', and go back to step 3.

(3.2) If not found , then

If rel-list includes R only, return failure (no connections found with name $cname_i$);

otherwise decrease t by 1, set rel-list to R , and go to step 2.

(4) (Find attribute A on the last connection name $cname_t$.)

Call reachc (curcname , found , R , c).

(4.1) If found add this connection to con-list , and traverse the connection to relation R' at the other end of the connection, set R to R'.

If reach (A, R) then return (R, con-list , f(con-list));

otherwise [if R is in rel-list , return failure; otherwise add R to rel-list and go back to step 4].

(4.2) (Failure.) If not found , return failure.

Auxiliary functions and procedures:

(1) function reach (A, R):

Local variables: $baserelist_1$, $baserelist_2$ both hold sets of relation names

If any attribute of R has name A, return TRUE ; otherwise do step (1).

(1) (Search for attribute A in the base relations of R .) Set $baserelist_1$ to the set of base relations of R , and repeat the following until $baserelist_1$ is empty: [set $baserelist_2$ to $baserelist_1$, set $baserelist_1$ to the empty set, for each relation R' in $baserelist_2$, do the following: [search if any attribute of R' has name A, if yes return TRUE and exit; otherwise put all base relations of R' in $baserelist_1$.]]. Return FALSE .

(2) function f(con-list , R_1, R_2):

If at least one connection in con-list in the direction from R_1 to R_2 is 1:N, return N; otherwise return 1.

(3) procedure reachc (curcname , found , R, c):

Local variables: *baserelist*$_1$, *baserelist*$_2$ both hold sets of relation names

If any connection c has name curcname in the direction *from* R , set found to TRUE and *return*; otherwise do *step* (1).

(1) (Search for connection c from the base relations of R .) Set *baserelist*$_1$ to the set of base relations of R , and repeat the following until *baserelist*$_1$ is empty: [set *baserelist*$_2$ to *baserelist*$_1$, set *baserelist*$_1$ to the empty set, for each relation R' in *baserelist*$_2$, do the following: [search if any connection c has name curcname in the direction from R, *if yes* set R to R', set found to TRUE , and return; otherwise put all base relations of R' in *baserelist*$_1$.]], set found to FALSE , return.

In the next section, we give the rules that are enforced on the names of attributes, relations, and connections, to ensure that the mechanism for connection traversal is not ambiguous.

4.3.7.2 Conditions for non-ambiguity of names

The conditions for *non*-ambiguity of an attribute specification in the algorithm of the previous section are now given. These conditions should be checked by the schema definition routines of the database management system. *All the conditions are easy to* check for.

(1) All relation names in a data model are unique.

(2) All attribute names in a relation are unique.

(3) All attribute names in all relations that are superrelations of a particular relation R_l at level *i*—and that have not appeared as attribute *names* in R_l or any superrelation of R_l at level j, $0 < j < i$—must be unique.

(4) All connection names in the direction from a relation are unique.

(5) All connection names in in the direction from all relations that are superrelations *of a particular relation R_l at level i—and that have not appeared as connection* names from R_l or any superrelation of R_l at level j, $0 < j < i$—must be unique.

Condition (1) is needed so that a name uniquely identifies a relation. Conditions (2) and (3) are needed so that the search for an attribute that is directly reachable from a relation will produce at most one attribute. Conditions (4) and (5) are needed so that at any given relation, the search for a connection with some given name which is reachable in the direction from that relation will produce at most 1 connection.

Note that although conditions (3) and (5) seem expensive to test, since we must carry out the test for all values of *i*, in most cases it will not be expensive. This is *because in most data models, it is quite unusual for* a relation to be a subrelation of more than one relation. Also, this procedure is carried out when the data model is being designed, and then only infrequently when new attributes, relations, and connections are introduced to the data model.

4.3.8 Scope rules and connection traversing in restricting logical expressions

Recall that when an attribute specification A_{lm} on relation R_l is specified via one or more connections, A_{lm} of R_l has the form

A of cname$_t$ of ... of cname$_1$ of R_l .

where A is an attribute of some relation R_e connected to R_l via a connection path. The *algorithm* in Section 4.3.7.1 gave the mechanism for locating R_e, and *for knowing whether the path defined* connects a single tuple or a set of tuples in R_e to a tuple in R_l.

If a set of tuples is specified, the language gives the option of restricting those tuples further by a logical expression. *In the specification of this* logical expression, *implicit references to a relation* R_n *is possible, where* R_n is specified as *follows.*

Suppose we have the following specification:

(A of cname$_t$ of ... of cname$_n$: <logical expression>)
of cname$_{n-1}$ of ... of cname$_1$ of R_l .

Then, when references are made within the <logical expression> to attribute or connection names without mention of a relation name, the relation R_n at the end of the connection path specified by the connection *name path* cname$_1$...cname$_n$ is implied. Hence, within that logical expression, the *focus* or *scope* changes from R_l to the relation R_n', formed of the subset of tuples in R_n connected to the tuple in R_l that is currently under consideration. The logical expression is applied to each of the tuples in R_n', and R_n' is further restricted to those tuples that satisfy the logical expression.

References in the logical expression to R_l still apply only to the tuple in R_l under *consideration. (Recall from* Section 4.3.3.2 that some tuples in R_l are selected by the WHERE clause, and for each of these tuples, the attribute values in the GET clause associated with that tuple are returned.)

The same rules apply to a restricting logical expression in the WHERE clause, only this time the rule applies to the tuple in R_l which is currently under consideration *for selection, rather than* already selected. An example is given in Section 4.3.9.

The only other case where scope changes is when a complete query is specified in the GET or the WHERE clause as a value from the database. All references within that query refer first to relations of the GET clause of that query. If a relation name is not found, then it is assumed to be a *reference* to a relation name in the outer query. This is consistent with standard scope rules in programming languages. If we do need to use two such different references to the same relation at the same scope level, we distinguish one occurrence from the other by attaching a *distinguished* character to all occurrences of one of them, a prime ('), say. This is also standard practice. If more than two occurrences are needed, which is quite unlikely, additional primes are appended to the relation name.

In general, if within one query Q1, a complete query Q2 exists, and within Q2 another query Q3 exists, then references to relation names within Q3 are first assumed to be to relation names specified in the GET clause of Q3. If no matching relation name is found, the relation names in the GET clause of Q2 are examined, and if no match is found, the relation names in the GET clause of Q1 are examined. If no match is found, then the query is ill-specified.

We do not allow implicit references to relation names in the WHERE clause of Q3 unless it is to the *(single) relation name specified in the GET clause of* Q3.

4.3.9 An example

We now give an example to illustrate the SMQL. Again, we use the model of Figure 15, which is duplicated here for convenience.

First, consider a retrieval request "Find the location where employee number 55 works". In the model of Figure 15, the LOCATION attribute is not in the EMPLOYEE relation but in the DEPARTMENT relation. This is so because when the data model was being designed, rules from the real-world situation were identified that specify that:

(a) every department has a single location, and

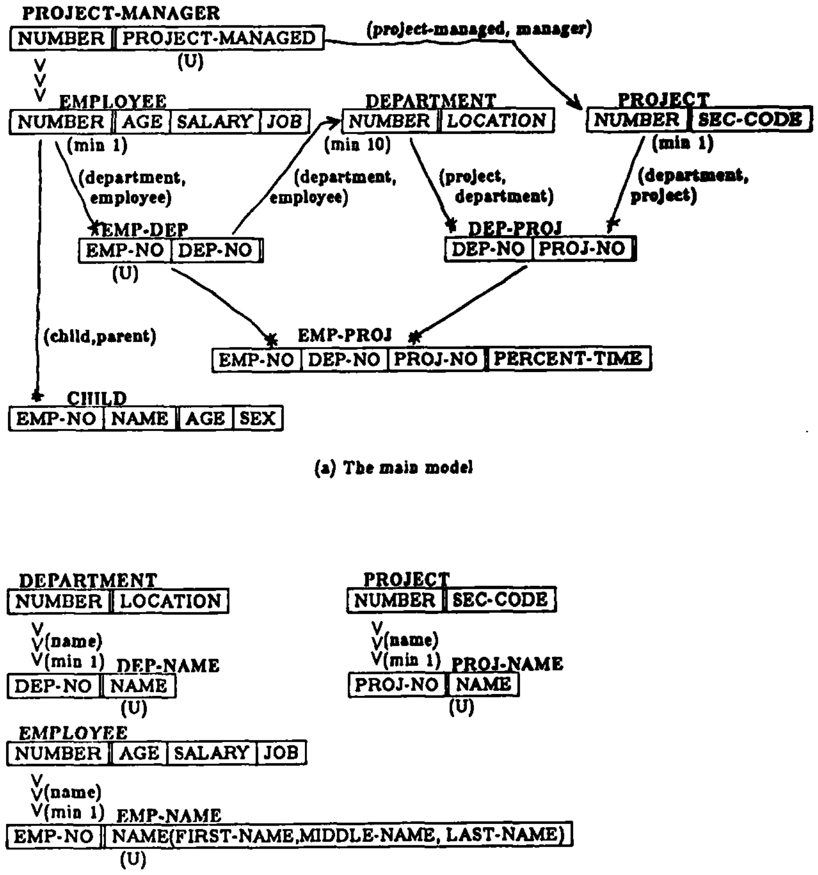
(b) an employee works in a single department.

These rules led to the design of the given model (see Chapter 5, where we discuss in detail the steps followed for the design of a structural model from an analysis of the real-world situation). These rules imply that the location of an employee is the location of his department. Hence, the query is transformed to "Get the location of the department of employee number 55" by the user, who is aware of the above rules, or can retrieve them by looking at the schema. To specify the *department* tuple that is connected to the desired employee tuple, we specify the appropriate connection names. We use the name of the connections from EMPLOYEE to DEPARTMENT, which is department in this case (see Figure 15). Note that two connections are traversed, from EMPLOYEE to EMP-DEP, and from *EMP-DEP to DEPARTMENT. However, only* one connection name is specified in the query, since the EMP-DEP relation is a connecting relation. The query is then:

Q2: GET LOCATION of department of EMPLOYEE
WHERE NUMBER of EMPLOYEE = *55*

The logical expression NUMBER of EMPLOYEE = *55* is satisfied by the employee tuple which has value for the attribute NUMBER = 55. The GET clause specifies the information about this employee to be retrieved—the location of his department. Since a single relation—EMPLOYEE—is specified in the GET clause, the relation specified in the WHERE clause must be the same. We can hence omit the relation name from the WHERE clause, so query Q2 can also be stated as shown in Q3.

Q3: GET LOCATION of department of EMPLOYEE
WHERE NUMBER = *55*



(a) The main model

(b) Lexicons of the model

Figure 15 Part of a structural data model

If the retrieval request of Q2 and Q3 is to be used frequently, it is more convenient to define a derived attribute LOCATION for EMPLOYEE as part of the data model by a derivation specification. This derived attribute can be seen as a means of predefining a frequently used operation. The derivation specification for LOCATION of EMPLOYEE is:

```
DEFINE DERIVED ATTRIBUTE LOCATION of EMPLOYEE TO BE
        GET LOCATION of department of EMPLOYEE
```

Now consider a slight variation of queries Q2 and Q3, as shown in Q4. Here, we want to retrieve the name and location of each employee who earns more than 40000 as salary. Here, the NAME attribute is retrieved from the specified EMPLOYEE relation, while the LOCATION attribute is specified as previously by a connection path from EMPLOYEE.

```
Q4: GET (NAME,LOCATION of department) of EMPLOYEE
    WHERE SALARY > 40000
```

Two differences exist in this query from the previous queries. First, more than one attribute value is retrieved for each employee tuple specified—the name and the location. Second, the WHERE clause specifies a set of employee objects rather than a single object as in the previous queries.

What we retrieve here is a relation of two attributes, NAME and LOCATION, and each tuple in that relation corresponds to an employee object that satisfies the condition of the WHERE clause (SALARY > *40000*). Here, both attributes are single-valued—NAME because it is an attribute of the EMPLOYEE relation, and LOCATION because the connection *from* EMPLOYEE to DEPARTMENT is N:1.

If the specified connection was N:M, instead of N:1, then all the department tuples connected to an employee tuple are specified via the department connection, and the set of all their locations is retrieved for each employee tuple. Hence, in this case we still retrieve a single tuple for each employee tuple, but the LOCATION attribute is now a repeating attribute of the resulting unnormalised relation.

Connections without names cannot be used for retrieval. They are only used to define structural integrity constraints. For example, the ownership connections from EMP-DEP to EMP-PROJ and from DEP-PROJ to EMP-PROJ in Figure 15 have no names. They are used to specify the constraint that an employee can work on a project only if his department is involved in that project. They are not used for retrieval.

Sometimes, it is useful to define additional retrieval paths by derived connections. No constraints are specified by these connections, and they are used for retrieval only. These derived connections serve to compress a path of 1:N connections—where the 1:N direction is the same for all the connections on the path—into a single 1:N connection. In the example of Figure 15, we need to define two derived 1:N connections to be able to connect employees and projects. One derived connection (project,employee) is defined from EMPLOYEE to EMP-PROJ and the second (employee,project) from PROJECT to EMP-PROJ. Both are ownership connections, since the paths were specified by two ownership connections. In general, if the path is formed entirely of ownership connections, the derived connection will also be an ownership connection. If one or more reference connections existed in the path, the derived connection will be a reference connection (see Section 6.4.1.6).

Now consider the query "Find the names, jobs and department numbers of all employees that work on project X, as well as the percentage of time each employee works on project X". We could attempt to express this query using the derived connections of the preceding paragraph (not shown in Figure 15) as follows:

```
Q5: GET (NAME, NUMBER of department, JOB, PERCENT-TIME of project)
        of EMPLOYEE
    WHERE X ∈ NAME of project of EMPLOYEE
```

Here, the WHERE clause uses the set element (∈) comparison because NAME of project of EMPLOYEE returns a set of values. This is easily deduced from the schema, since the connection from EMPLOYEE to EMP-PROJ is 1:N, so a set of project tuples are connected to each employee tuple, and hence a set of project names is returned for each employee. Hence, we are here comparing a single value with a set of values, and the appropriate operators are ∈ and ∉.

We also allow the comparison of single values using set operators. In this case, the single value is *transformed* to the set that contains only that value. Hence, the query Q5 could also be expressed as shown in Q6.

```
Q6: GET (NAME, NUMBER of department, JOB, PERCENT-TIME of project)
        of EMPLOYEE
    WHERE NAME of project of EMPLOYEE ⊇ X
```

Note that in this example, an equality operator (=) in place of ⊇ expresses a different condition, and would return employees that work only on project X since it is set equality.

In Q5 and Q6, the value of PERCENT-TIME of project of EMPLOYEE is located by tracing the (project,employee) derived connection (not shown in Figure 15) from EMPLOYEE to EMP-PROJ, where the attribute PERCENT-TIME is located. Once an attribute is located on the connection path, no further traversing is necessary, since the rule is that the first encountered attribute that satisfies the specification is the one retrieved (see Section 4.3.7.1).

On a closer examination of query Q5, we see that the WHERE clause specifies a set of tuples from EMPLOYEE. The GET clause specifies the retrieval of the following attributes for each employee tuple in the set:

(1) The NAME and JOB attributes from the EMPLOYEE relation. A single value is retrieved for each employee tuple.

(2) The attribute NUMBER from the tuple in the DEPARTMENT relation that is connected to a specified employee tuple. Since the connection in the schema is N:1, a single department name is retrieved for each employee tuple.

(3) The attribute PERCENT-TIME from the tuples connected to each employee tuple *in the relation EMP-PROJ. Here, a set of values is retrieved for each employee* tuple since the connection is 1:N.

Hence, we do not get the answer to our original requested query if it was intended that for each employee, only the value of PERCENT-TIME for project X be retrieved.

Whenever an attribute in the GET clause is to return a set of values, we should have the ability to further restrict that set. Hence, we can express the correct query as shown in Q7.

Q7: GET (NAME, NUMBER of department, JOB, PERCENT-TIME of project:
NAME = X) of EMPLOYEE
WHERE NAME of project of EMPLOYEE $\supseteq X$

In general, whenever an attribute specifies a set of values rather than a single value, we can restrict that set. We accomplish this by specifying the tuples from which we retrieve the values via a logical expression, identical to the logical expression that specifies tuples in the WHERE clause.

However, this logical expression is applied to the relation formed from the subset of tuples in the PROJECT relation which are connected to the employee tuple under consideration. The traversing of connections specified in that logical expression (none are specified in our example) start at the PROJECT relation.

We can also express the query of Q7 by starting from the PROJECTS relation. This query is shown in Q8. In this case, the WHERE clause restricts the selection of tuples from the PROJECT relation, and the qualifier following the PERCENT-TIME attribute restricts the set of values to the ones associated with project X.

Q8: GET (NAME, JOB, PERCENT-TIME of project: NAME = X, NUMBER
of department) of employee of PROJECT
WHERE NAME of PROJECT = X

Technically, Q8 retrieves a single tuple for project X, which is composed of a compound repeating attribute that includes all the attributes to be retrieved. This is so because the PROJECT relation is the one specified in the GET clause.

When no WHERE clause is specified, WHERE TRUE is implied, so all the tuples in the relation of the GET clause are selected. For example, the query "Give me the department number and the project numbers that the department participates in for *all known departments" is expressed as shown in* Q9.

Q9: GET (NUMBER, NUMBER of project) of DEPARTMENT

Next, we illustrate the use of the set operators. The query "Give me the department numbers of all departments that are associated with at least two of the projects named X, Y, or Z" would be specified as follows:

Q10: *GET NUMBER of DEPARTMENT*
WHERE COUNT (NAME of project of DEPARTMENT $\bigcap\{X, Y, Z\}) \geq 2$

Here, to test that two sets have at least two elements in common, we count the number of elements in the intersection of the two set, and use that number as a single numeric value in a comparison operation.

To demonstrate the orderings that apply to a low-level domains of type STRING, consider the query "Retrieve the names of employees in the payroll department whose name starts with any of the letters A to J, and order the output alphabetically". The query Q11 carries out this request.

Q11: GET NAME of EMPLOYEE ORDER BY NAME
WHERE $SAL > 30000$ AND NAME of department = *PAYROLL* AND NAME
BEFORE K

The domain of dates is also subject to the obvious total order, with the more recent dates considered greater in value. For dates, we also use the operators BEFORE and AFTER for $<$ and $>$, respectively.

Next, an example to demonstrate that the explicit values which are used in a comparison need not be known values—they could be values that are retrieved from the database by another query. For example, if in query Q11 we wanted to retrieve all employee names *for those employees that work on at least one of the projects on which* employee number *3120* works (rather than a set of projects we know explicitly as in Q11), we can write:

Q12: GET NAME of EMPLOYEE
WHERE COUNT (NUMBER of project $\bigcap$ (GET NUMBER of project of
EMPLOYEE WHERE NUMBER of EMPLOYEE = *3120*)) ≥ 1

Here, the inner query is completely independent from the main query. Hence, any occurrence of EMPLOYEE within the inner query is independent from the reference to EMPLOYEE in the outer query. This is consistent with standard scope rules in programming languages. If we do need to use two such different references to the same relation at the same scope level, we distinguish one occurrence from the other by attaching a distinguished character to all occurrences of one of them, a prime ('), say. This is also standard practice. If more than two occurrences are needed, which is quite unlikely, additional primes are appended to the relation name.

A simpler example of the use of values retrieved from the database follows. In this example, we assume that a reference connection (manager, supervisees) exists from EMPLOYEE to EMPLOYEE (not shown in Figure 15), and that it is of cardinality N:1 (an employee has only one manager). The query is "Give me all employee names for employees who earn more than their managers". In this query, a single value comparison is used, since an employee has only one manager, and hence only one salary of manager.

```
Q13: GET NAME of EMPLOYEE
     WHERE SAL > SAL of manager
```

Query Q13 is different than Q12 in that the values from the database that are used in selecting the appropriate employee tuples are both values of attributes connected to the particular employee tuple being checked for satisfying the logical expression of the WHERE clause—his salary, and the salary of his manager.

The following queries illustrate the use of functions. First, consider functions that apply to numeric domains. Consider the request "Retrieve the average age and standard deviation of the ages of all employees". We write:

```
Q14: GET (AVERAGE, SD) AGE of EMPLOYEE
```

Here, the average is computed in the customary manner. If, on the other hand, we want to retrieve the average salary of employees, say, such that each salary value appears once in the computation of the average, then we first create a derived relation EMP-SALARIES of one attribute SALARY.

DEFINE DERIVED RELATION EMP-SALARIES TO BE GET SALARY of EMPLOYEE

The derived relation is the set of unique salary values that appear in the EMPLOYEE relation. We can calculate their average as shown in Q15. Note that it is unusual to request the average *in this manner*.

```
Q15: GET AVERAGE SALARY of EMP-SALARIES
```

We can also calculate the average on a restricted multiset. For example, to get the average salary of employees who work in the *PAYROLL* department, we write:

```
Q16: GET AVERAGE SALARY of EMPLOYEE
     WHERE NAME of department of EMPLOYEE = PAYROLL
```

To retrieve the name of each department, and the average salary of its employees, we write:

```
Q17: GET (NAME,AVERAGE SALARY of employee) of DEPARTMENT
```

The other standard function that we have is COUNT, which returns the number of values in a specified set, and is applicable to a set of any domain. The COUNT function

may also be specified on a connection name only, rather than an attribute, in which case it returns the number of tuples connected to a specified tuple via that connection. For example, to retrieve the name and number of employees in each department, we specify:

```
Q18: GET (NAME, COUNT employee) of DEPARTMENT
```

The COUNT function can also be specified over a relation name only, in which case it returns the number of tuples in that relation that satisfy the conditions of the WHERE clause. For example, to retrieve the number of employees that earn less than 15000, we write:

```
Q19: GET COUNT EMPLOYEE
     WHERE SALARY < 15000
```

The functions MAX and MIN return the maximum and minimum values from a multi-set. For example, to retrieve the employee (or set of employees) that make the highest salary, and the name of their department(s), we write:

```
Q20: GET (NAME, NAME of department) of EMPLOYEE
     WHERE SAL = GET MAX SALARY of EMPLOYEE
```

The functions MAX and MIN apply to any ordered domain. MAX returns the highest value in the order from a multi-set of values that it is applied to, and MIN returns the lowest value in the order. We can also use these functions to return the i^{th} largest or smallest element, by placing an (i) following the operator. Here, we also take duplicates into account. For example, to return the name of the fifth highest paid employee, we write:

```
Q21: GET NAME oi EMPLOYEE
     WHERE SALARY = GET MAX (5) SALARY of EMPLOYEE
```

Note that either of the queries Q21 and Q22 may return a single tuple or a set of tuples.

Next, consider an example of the EXISTS function. Consider the query "Are there any employees that are not associated with a project". This could be stated as:

```
Q22: GET EXISTS EMPLOYEE
     WHERE NOT EXISTS project of EMPLOYEE
```

This query illustrates the use of the EXISTS function in both the GET and the WHERE clause. In the WHERE clause, for each employee tuple, the function will return TRUE or FALSE depending on whether the tuple is connected to at least one project tuple or not. This logical expression is identical to

COUNT project of EMPLOYEE = 0 ,

so the use of the EXISTS function in the WHERE clause is redundant.

In the GET clause, the function will return Yes if at least one employee tuple satisfied the condition of the WHERE clause, and No otherwise. Note that in this example, EXISTS was applied to a *relation name in the GET clause, and to a connection* name in the WHERE clause. It can also be applied to attribute names.

Queries *of this type* can sometimes be answered by *looking only* at the schema. For example, in Figure 15, the schema specifies that every employee is associated with at least one department. This is so because the relationship EMPLOYEES:DEPARTMENTS is a dependency of EMPLOYEES on DEPARTMENTS, and the cardinality is *N*:1. Hence, if we ask the query "Do any employees work in more than one department", say, we could answer No by only looking at the schema, without having to access the database.

This is also true of retrieval requests when the condition of the WHERE clause can be evaluated to FALSE without accessing the database. In this case, the query can return an empty response directly.

The computation of functions over sets of unique values, as *illustrated* by query Q15, is one possible application of using derived structures in the query language. *Another use is when we want to group tuples together* by value of a certain attribute. If that attribute was already defined as a connecting attribute to another relation in the data model, we would have no problem. For example, in Figure 15, it is straightforward to group employees by department. However, in order to group employees by salary, say, we need to define the derived relation EMP-SALARIES, and in addition we need to define a derived connection from EMPLOYEE to EMP-SALARIES called (salary, employees), say.

```
DEFINE DERIVED CONNECTION (salary, employees) TYPE DIRECT REFERENCE
       FROM EMPLOYEE (SALARY) TO EMP-SALARIES (SALARY)
```

Now, we can retrieve each salary, and the names of employees who earn that salary using the derived connection as follows:

```
Q23: GET (SALARY, NAME of employees) of EMP-SALARIES
```

So far, all our examples have had a single relation name *in the GET* clause. For the majority of cases, we will not need to specify more than one relation. Now let us *consider cases where we need to specify more* than *one* relation name in the GET clause.

Connections exist in a data model to represent all known semantic connections between relations that specify structural integrity constraints. However, sometimes we want to relate tuples from different relations in a query based on connections that do not have a semantic significance, and hence that are not defined in the model. For example, the query "Retrieve all employee and department name pairs for employees and departments that have the same number". Semantically, it makes no sense to relate department numbers to employee numbers (unless some rule exists that relates the two, but we assume there is no such rule). However, one might still want to pose the above query (for trivia purposes, say). The query is given in Q24.

```
Q24: GET NAME of EMPLOYEE, NAME of DEPARTMENT
     WHERE NUMBER of EMPLOYEE = NUMBER of DEPARTMENT
```

Query Q24 corresponds to performing an explicit JOIN operation between two relations. In all the previous queries, the joins were specified simply by using the connection names, since they were predeclared via connections that represented semantic aspects of the data model. In this case, each pair of tuples, one from each relation, are tested to see if they satisfy the WHERE clause, and if so, a tuple in the output is created with the desired attribute values as discussed in Section 4.3.3. However, we cannot omit the relation name from attributes specified in the WHERE clause any longer, since it could be either of two relations.

The absence of a WHERE clause in this case creates a cross-product of the tuples, since all pairs of tuples will satisfy the assumed TRUE condition. Note also that a JOIN on the same relation can be performed using a primed relation name for one of the occurrences of the relation.

Next, consider examples of update requests. First, consider a request to change the value of an attribute. The request "Change the salary of employee number 632 to 25000 dollars" is expressed as:

```
Q25: CHANGE SALARY of EMPLOYEE TO 25000
     WHERE NUMBER = 632
```

We can also change an attribute from a tuple in a connected relation. For example, referring to Figure 15, we can change the department of a particular employee to the *Payroll* department as follows:

```
Q26: CHANGE DEP-NO of department of EMPLOYEE TO (GET NUMBER
         of DEPARTMENT WHERE NAME = PAYROLL)
     WHERE NAME = (JOHN,PAUL,SMITH)
```

Note that the preceding update request is well specified with respect to the model *of Figure 15. It only changes the* DEP-NO attribute in the EMP-DEP relation, but does not change the NUMBER attribute of the *PAYROLL* department (recall the rules for specifying attributes of Section 4.3.7). That is why it is useful to give connecting attributes different names in different relations.

The new value of an attribute, which is specified following the keyword TO in the CHANGE clause, can be specified using any valid way of specifying a value, including a retrieval from the database, as in the preceding example.

We can also specify an attribute in a connected relation that is a repeating attribute, because of a 1:*N* connection. In this case, we must specify a set of values to replace the current set. However, using the $\bigcup$ operator, we can augment a set by a new value, say. For example, if we want to associate project number *10* with all departments located in *Alexandria*, in addition to the current projects that each department is associated with, we can write:

```
Q27:CHANGE PROJ-NO of project of DEPARTMENT TO (PROJ-NO of
        project of DEPARTMENT ∪ {10})
    WHERE LOCATION of DEPARTMENT = Alexandria
```

This kind of update is very difficult to implement efficiently. It is preferably specified by explicit insertion of the appropriate connecting tuples in the DEP-PROJ relation.

Next consider insertions. The simplest insertion is to insert a tuple in a relation. For example, to insert a new employee tuple in the EMPLOYEE relation, we write:

```
Q28:INSERT EMPLOYEE: < 2314, 55, 30000, Engineer >
```

The values of the new tuple correspond to the attributes NUMBER, AGE, SALARY, and JOB, respectively, and are specified in the same order as in the schema. Alternatively, the values of the tuple could be specified as pairs of an attribute name and an attribute value.

An insertion that violates the constraints of the data model should be prohibited by the database management system. Hence, the above insertion would not be permitted on the model of Figure 15, since the constraint specified by the identity connection from EMPLOYEE to EMP-NAME is violated by this insertion. We cannot even insert the tuple that includes the name of the employee in EMP-NAME first, since a (min 1) constraint is specified on the identity connection, which means that tuples in EMPLOYEE and EMP-NAME must be in a one-to-one correspondence. Hence, we need a capability to insert several tuples in an insertion transaction, and check that the constraints have not been violated at the end of the complete transaction, even though temporary violations may have occurred.

An insertion transaction is defined by a name, a parameter list, and the operations carried out as part of a transaction. It usually involves insertion of tuples in more than one relation. For example, to insert a new employee, we need to insert not only the values for the attributes specified in the EMPLOYEE relation, but also his department (by inserting a tuple in EMP-DEP), his name (by inserting a tuple in EMP-NAME), and possibly the set of projects he will work in (by inserting some tuples in the EMP-PROJ). We can define a transaction called new-emp, and invoke it whenever a new employee is added to the database.

Now a check for consistency is made after the complete transaction, and a transaction is allowed if consistent upon terminating all the insertion operations even though temporary integrity violations may have occurred. We can also define update transactions that include DELETE and CHANGE statments.

An example of a transaction follows.

```
DEFINE TRANSACTION new-emp(FIRST-NAME, MIDDLE-NAME, LAST-NAME,
        NUMBER, AGE, SALARY, JOB, DEPT-NUMBER,
        { < PROJ-NUMBER, PERCENT-TIME > }):

        INSERT EMP-NAME: < NUMBER, FIRST-NAME, MIDDLE-NAME,
            LAST-NAME >
        INSERT EMPLOYEE: < NUMBER, AGE, SALARY, JOB >
        INSERT EMP-DEP: < NUMBER, DEPT-NUMBER >
        INSERT EMP-PROJ: { < NUMBER, DEPT-NUMBER, PROJ-NUMBER,
            PERCENT-TIME > }
END
```

To delete a tuple, the relation name is specified, and the WHERE clause specifies the tuples to be deleted. For example, to delete the employee *JOHN PAUL SMITH*, we write:

```
Q29:DELETE FROM EMP-NAME
    WHERE NAME = (JOHN, PAUL, SMITH)
```

The deletion of a tuple will automatically result in the deletion of all tuples owned by it in other relations in the database. For example, all information specified in the model to be properties that describe *JOHN PAUL SMITH* are deleted by Q29.

Hence, Q29 will not only delete the designated tuple from the relation EMP-NAME, but will also delete tuples from each of the relations EMPLOYEE, EMP-DEP, and EMP-PROJ that represent the deleted employee, automatically. This is because of the nature of the identity an ownership connections (see Section 3.3).

4.3.10 Summary

We presented in this section a proposal for a Structural Model Query Language (SMQL), an object-oriented query language for the structural model. The language has some unique features as well as features that are available in many query languages.

The attribute traversing mechanism provides us with a functional specification of attributes that are connected to a relation, and hence is similar to functional languages. However, the separation of a query into two clauses in a formal, non-ambiguous query language is a new concept.

Other new features are the set-oriented operators of the logical expressions, and the way we can restrict repeating attributes which are to be retrieved by specifying separate logical expressions. Also, few languages have powerful derivation facilities as proposed for the SMQL.

On the other hand, most languages have similar update capabilities, and capabilities for defining attributes with domain constraints. Many query languages provide the arithmetic and functions we covered in Section 4.3.6.

5 DATA MODEL DESIGN

5.1 INTRODUCTION

In this chapter, we show how the real-world structures discussed in Chapter 2 can be correctly represented using the structural model. Recall that in Chapter 2, we categorised the aspects of the real-world that are important to database modelling. The concepts we wanted to represent can be formally described as follows:

(a) Object classes: An object class O is a class of objects $\{o_1, o_2, \ldots\}$ of identical structure that correspond to real-world objects. Objects in the class O are created and destroyed to correspond to the objects that exist in the real-world. An object class O is specified by an ordered set of properties $\{p_1, \ldots, p_n\}$. Each property p_i of the object class is associated with a domain of values $\{v_{i_1}, v_{i_2}, \ldots\}$, and in addition may be specified to be unique, repeating, optional, or compound. An individual object o_i is a tuple of data values $\langle i_1, \ldots, i_n \rangle$ where data value i_k corresponds to property p_k. Data value i_k is either a single value, a tuple of values, a set of values, or a set of tuples of values. The default is a single value from the domain of p_k. If p_k is specified to be compound, then p_k must be further specified by m properties $p_{k_1}, \ldots, p_{k_m}$, each of which may be further specified. The domain of p_k is then the cross product of the domains of $p_{k_1}, \ldots, p_{k_m}$. The data value i_k in this case is a tuple of values, which is essentially another object. If property p_k is specified to be repeating, then item i_k is a set of values from the domain of p_k, or a set of tuples if p_k was compound. If p_k is specified to be unique, then each value from the domain of p_k can appear in at most one object of the class at any given moment. If p_k is optional, then item i_k can have a missing value, not from the domain of p_k.

(b) Relationships: A relationship is a mapping between two object classes, or two categories of object classes. A category is a super class of objects that includes all objects of several object classes. A relationship between two object classes P and Q maps each object o_{P_i} in class P to a set of objects in class Q, possibly none. A relationship is constrained by its structural properties, which restrict the possible mappings of objects. Structural properties are the cardinality and the dependency, and are discussed fully in section 2.2.

(c) Subclasses: A subclass R of an object class O is a subset of the objects from O that are described by additional properties, or participate in separate relationships, and hence are distinguished from other objects in O.

The above simplified view of the world is what a user would expect to see. The structure of an object class as seen by one user is basically hierarchical. For example, if the user is mainly interested in the object class of EMPLOYEES, he may consider a department and its properties to be properties of the EMPLOYEES object class, since the user might want to access information about the department that an employee works in directly as properties of the employee object. Similarly, if the user at some other time is mainly interested in information concerning departments, he might consider the set of employees that work in each department as a compound repeating property of that department.

The problem with the above two views is that both are correct. Hence, we want to be able to present the user with either of the above views. However, if we literally represent both views, then the representation becomes incorrect. This is so because we have represented information about employees and about departments redundantly—once in the EMPLOYEES object class, and another time in the DEPARTMENTS object class—while the information in actual fact is the same. Hence, what we need are mechanisms that represent the object classes and their properties in a *non-redundant* manner to ensure consistent data, yet provide the user with a view that may be *highly redundant*.

We also want to ensure the correct representation of structural properties of relationships, since these properties are rules that apply in the real-world. Hence, if the rules are violated in the database, it implies either that data is being introduced that is erroneous, or that the rules that have applied in the real-world have changed. In either case, an early indication of the error, or of the change in real-world rules, is important.

When relationships are between categories rather than object classes, we should still be able to represent the relationship, and its structural properties, correctly. However, these cases do not occur very frequently in real-world situations.

Finally, we want our model to correctly represent the properties that are expected of a subclass. Since objects in a subclass are also objects of the base object class, an object of the subclass shares all the properties that are of the main class.

In this chapter, we will discuss how to represent these above concepts. A particular emphasis is placed on the *correct* representation of the behaviour of the model. A data model is not just a grouping of data items together for convenient access. Rather it is a *representation* of the real-world, and the constraints that apply in the real-world, as much as possible. A data model is also expected to provide a user with the capabilities to interact with the database according to his current concepts, not according to the way the data items have been partitioned and stored. What we try to demonstrate here is that these goals are achievable to a large extent if we place more effort in the analysis of the real-world situation, and in the design of the data model.

In Section 5.2, we discuss representation of object classes and their properties. In Section 5.3, we discuss the representation of relationships, their structural properties, and the methods of maintaining these structural properties. In Section 5.4 we discuss the representation of other semantic concepts, such as subclasses. In Section 5.5 we discuss the choice of names for relations, attributes and connections, so that the user interaction with the database, using the Structural Model Query Language, will be close to his or her perceptions. Finally, in Section 5.6 we discuss in detail the steps to be taken in designing a data model, and consider an unusually complex data model to illustrate the design steps.

5.2 REPRESENTATION OF OBJECT CLASSES

An object class is represented by one or more connected relations. One of these relations, which we call the *main* relation for the object class, will have the same name as the object class, or some close variation of it. Simple properties, including only one identifying property (or set of identifying properties) are represented as attributes of this main relation. Other relations connected to the main relation are used to represent repeating properties of the class, and duplicate identifying properties.

Repeating properties are represented as a nest relation owned by the main relation. Hence, an ownership connection exists from the main relation to every relation that represents repeating properties of the object class.

By judicious choice of connection names, the repeating properties can be accessed in the SMQL just as any other properties of the class. In addition, the ownership connection represents the correct update constraints: when the tuple that represents an object is deleted, the tuples that represent its repeating properties are automatically deleted.

Duplicate ruling parts are represented by lexicon relations, connected by an *identity connection from the main relation to the lexicon*. The attributes in the lexicon are automatically referenced by the structural model query language, since the connection is an identity connection (see Section 4.3.7). Since a 1:1 correspondence should exist between tuples in the lexicon and tuples in the main relation, a (min 1) should be specified on the identity connection.

Consider the EMPLOYEES and DEPARTMENTS object classes of Figure 2 of Chapter 2. They can be represented in the structural model as shown in Figure 16. The choice of a single ruling part is important because the ruling part is used to define connections of this relation with other relations. If duplicate ruling parts exist in the main relation, some connections may use one ruling part and others a second, which leads to undue complexity. No additional difficulty is created by introducing the lexicon relations, since the attributes in the lexicon are accessed by the SMQL in the same way as the primary ruling part. In addition, in most implementations, they would be stored in the same file as the other attributes.

Note that the properties DEPARTMENT and PROJECTS of EMPLOYEES, and PROJECTS of DEPARTMENTS (Figure 2, Chapter 2) are relating properties, and are not shown in Figure 16. These aspects will be shown in Section 5.3, which is devoted to modelling relationships.

First consider the two lexicon relations. The (min 1) constraints specified on the identity connections from EMPLOYEE and DEPARTMENT mean that every tuple in the lexicon must be connected to a tuple in the corresponding main relation. Hence, one-to-one correspondences exist between tuples in EMPLOYEE and EMPLOYEE-NAME, and also between tuples in DEPARTMENT and DEPARTMENT-NAME.

relation name: EMPLOYEE (main relation for EMPLOYEES object class)
ruling part: NUMBER
dependent part: ADDRESS:(STREET, NUMBER, APT-NO, CITY, STATE, ZIPCODE), BIRTH-DATE:(MONTH, DAY, YEAR), AGE, SEX, SALARY, JOB-NAME

relation name: EMPLOYEE-NAME (lexicon to represent duplicate identifying property EMPLOYEE-NAME)
ruling part: EMP-NO
dependent part: NAME:(FIRST-NAME, MIDDLE-NAME, LAST-NAME)

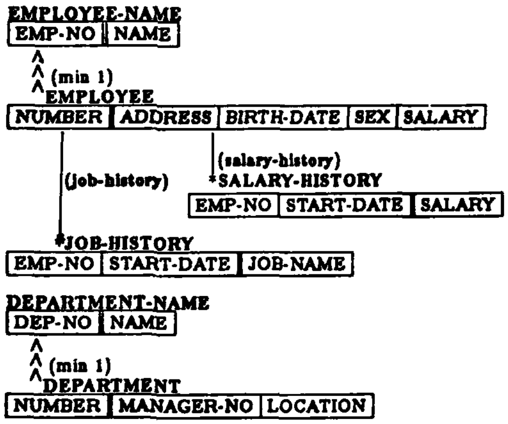
relation name: SALARY-HISTORY (nest relation to represent SALARY-HISTORY)
ruling part: EMP-NO, START-DATE:(MONTH, DAY, YEAR)
dependent part: SALARY

relation name: JOB-HISTORY (nest relation to represent JOB-HISTORY)
ruling part: EMP-NO, START-DATE:(MONTH, DAY, YEAR)
dependent part: JOB-NAME

relation name: DEPARTMENT (main relation for DEPARTMENTS)
ruling part: NUMBER
dependent part: MANAGER-NO, LOCATION

relation name: DEPARTMENT-NAME (lexicon for DEPARTMENT-NAME)
ruling part: DEP-NO
dependent part: NAME

(a) The object classes EMPLOYEES and DEPARTMENTS and their properties



(b) Graphical Representation

Figure 16 Representing properties of object classes

Now consider access to the EMPLOYEE-NAME relation in the Structural Model Query Language (SMQL, see Section 4.3). It suffices to specify NAME of EMPLOYEE, since the connection *from* EMPLOYEE to EMPLOYEE-NAME is an identity connection. Whenever an identity connection exists, and an attribute cannot be located in a relation, its base relations are searched for that attribute name (see Section 4.3.7). Hence, the attribute NAME is located in the relation EMPLOYEE-NAME, and its value can be accessed in the database for any specified employee tuple as though it were a *property of the* EMPLOYEE *relation*.

The repeating properties of the object class EMPLOYEES are represented by the nest relations JOB-HISTORY and SALARY-HISTORY, owned by the EMPLOYEE relation. These nest relations are accessed using the connection names. For example, consider the retrieval request "Retrieve the jobs that *Bob Smith* has held". The corresponding SMQL query is:

```
Q30: GET JOB of job-history of EMPLOYEE
     WHERE FIRST-NAME = Bob AND LAST-NAME = Smith
```

Note the use of the simple attributes FIRST-NAME and LAST-NAME of the compound attribute NAME of EMPLOYEE. They are accessed as any other attribute. Hence, the only difference a compound attribute makes is that it allows direct reference to the set of attributes that form the compound attribute.

The ownership connections in Figure 16 have only one name each, in the direction from the main relation to the owned nest relation. This is because access to the attributes in the owned nest relations starts at the main relation since they represent properties of the object class that the main relation represents. If it is expected that access will be needed in the reverse direction, a second name must be given to the connections.

The attributes that are placed in the main relation and connected relations which represent an object class are those that are directly dependent on the object of that class. In general, other attributes exist which the users would like to access as attributes of that class, but which are actually attributes of another object class. For example, one might want to refer to the LOCATION of an EMPLOYEE. If we assume that in a particular company, an employee works in one department, and a department is located in a single place, then the LOCATION attribute is part of the DEPARTMENT relation. However, a relationship will be represented to connect an employee to his department, as discussed in the following section. If the connection is as shown in Figure 18, we can define a derived attribute LOCATION of the EMPLOYEE relation, using the derive statement of the SMDL and SMQL (see Section 4.3.5).

5.3 REPRESENTATION OF RELATIONSHIPS

As we saw in Section 2.2.1, a relationship between more than two object classes can always be decomposed into several binary relationships between *only* two object classes. Hence we will only consider binary relationships here.

In this section, we show how a relationship of known cardinality and dependency properties can be represented by the connections of the structural model. In particular, we show that the complete set of structural properties can be represented, and that the constraints specified by connections between relation will reject transactions on the database that cause the violation of a structural property.

In Section 5.3.1, we show the different ways a relationship can be represented in the structural model, and in Section 5.3.2, we show the different techniques for specifying how the structural properties can be maintained, and how the structural model can represent these techniques.

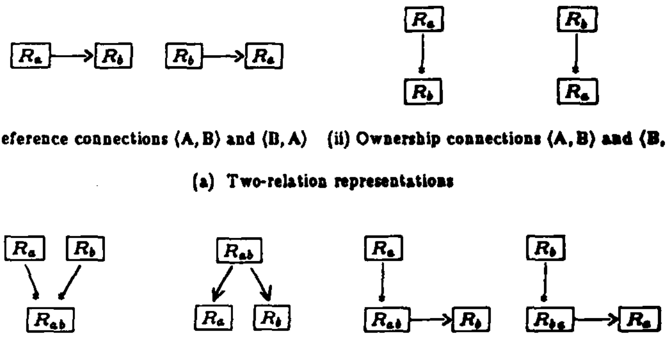
5.3.1 Binary relationships between object classes in the structural model

A relationship between two object classes can be represented using five constructs in the structural model. Two of these constructs use two relations—which are the main relations that represent the two object classes—and a connection between the two relations. The connection may be a reference connection or an ownership connection. The other three representations use three relations, two of which are the main relations that represent the two object classes, and the third relation is a connecting relation. The connecting relation may be an association, a nest (of references), or a primary relation.

Figure 17 shows the different representations between two object classes called A and B. In our diagrams, the boxes labeled R_a and R_b are the main relations that represent the object classes A and B, respectively, while the boxes labelled R_{ab} represent *auxiliary connecting relations*. Each distinct representation for a binary relationship in the structural model is given a name for easy reference. The two object classes A and B can be of any semantic type, including relationship classes.

Two of the representations—the association and the primary— are symmetric with respect to R_a and R_b. The other three representations—reference, ownership, and nest *of reference*— are not symmetric with respect to R_a and R_b. The names chosen for each representation are the names of the connection in two-relation representations, and the names of the auxiliary connecting relation for the three-relation representations.

In subsequent diagrams, when we represent an object class, we will only show its main relation, the ruling part attributes of the main relation, and any connecting attributes that may be in the main relation. The ruling part represents a single identifying property, or a set of identifying properties, and the connecting attributes represent the relating properties of the object classes when we have a two-relation representation. In a three-relation representation, the connecting attributes are in the auxiliary connecting relation. All other properties of an object class can be represented as discussed in Section 5.2, but will not be shown since they are not relevant to representing relationships.



(i) Reference connections (A, B) and (B, A) (ii) Ownership connections (A, B) and (B, A)

(a) Two-relation representations

(i) Association (ii) Primary (iii) Nest of references (A, B) and (B, A)

(b) Three-relation representations

Figure 17 Representing a relationship A:B in the structural model

5.3.1.1 Two-relation representations

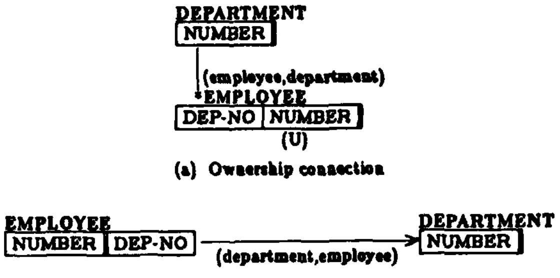
Only relationships of cardinality 1:1 or 1:N can be represented by two connected relations. An ownership connection (A,B) and a reference connection (B,A) represent a partial dependency of B on A. Figure 18 shows a 1:N relationship DEPARTMENTS: EMPLOYEES as represented by these constructs. In both representations, the main relation EMPLOYEE includes the connecting attribute DEP-NO. We assumed here that employee numbers are unique within the whole organisation.

For the nest representaticn (Figure 18a), the 1:N cardinality is enforced by specifying the attribute NUMBER of EMPLOYEE to be unique. Hence, each tuple in the EMPLOYEE relation represents a unique employee. For the reference representation, this uniqueness is implicitly specified since the attribute NUMBER of EMPLOYEE is the ruling part.

In Figure 18, the partial dependency of EMPLOYEES on DEPARTMENTS is implicitly specified by inclusion of the (mandatory) connecting attribute DEP-NO in the EMPLOYEE relation. Recall that we do not allow optional connecting attributes (refer to Definition 4, Section 3.2.2).

To represent a 1:1 cardinality, the connecting attribute must be restricted to unique valuer. Figure 19 shows an example, a partial dependency of MANAGERS on DEPARTMENTS, and of cardinality 1:1. The connecting attribute DEP-NO of the MANAGER relation is *restricted* to be unique to enforce the 1:1 cardinality.

A partial dependency (*i*) of DEPARTMENTS on EMPLOYEES cannot be repre-



(a) Ownership connection

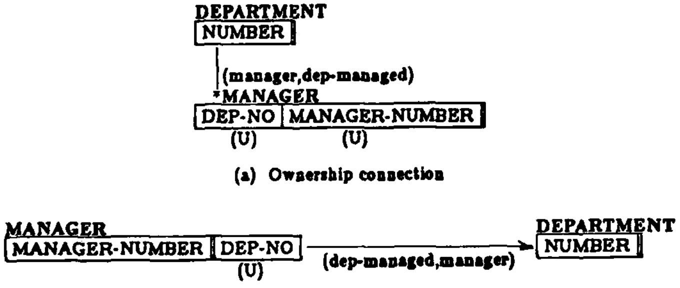
(b) Reference connection

Figure 18 Partial dependency of EMPLOYEES on DEPARTMENTS of cardinality N:1

sented by the two-relation representation, since this would require an optional connecting attribute *DEP-NO* of *EMPLOYEE*. This case is well represented by three relations; the third connecting relation holds the dependency information.

To specify a total dependency (i) relationship DEPARTMENTS:EMPLOYEES, a (min i) is attached to the ownership or the reference connections of Figure 18. For a 1:1 relationship (Figure 19), only a (min 1) makes sense, and then a 1:1 correspondence is specified. For the 1:N relationship, any i can be specified. If a (max j) is specified on the connection, then $i \leq j$.

The difference between the ownership and reference connection is how the partial dependency rule is enforced in case of an update that will cause an inconsistency to



(a) Ownership connection

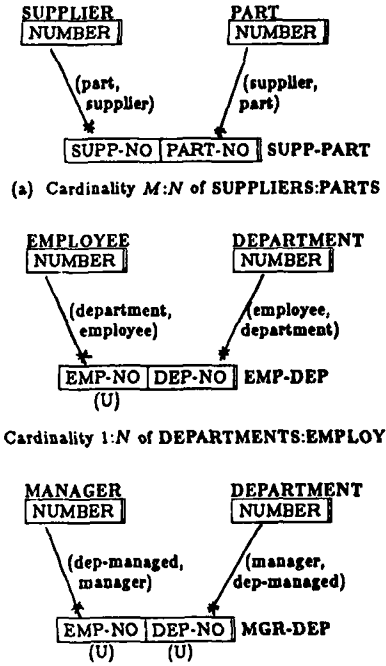
(b) Reference connection

Figure 19 Partial dependency of MANAGERS on DEPARTMENTS of cardinality 1:1

the database. For example, in Figure 18, if an attempt is made to delete a department tuple that is connected to one or more employee tuples in the database, this update will violate the partial dependency constraint. The reference connection will not allow such a deletion to happen. The ownership connection carries out the deletion and automatically deletes all employee tuples connected to the department tuple, so that the update becomes consistent. We will discuss these and other rules for integrity maintenance, and how to choose between them in Section 5.3.2.

5.3.1.2 Three-relation representations

Now consider the three-relation representations. Without any additional con-



(a) Cardinality M:N of SUPPLIERS:PARTS

(b) Cardinality 1:N of DEPARTMENTS:EMPLOYEES

(a) Cardinality 1:1 of DEPARTMENTS:MANAGERS

Figure 20 No dependency relationships of different cardinalities

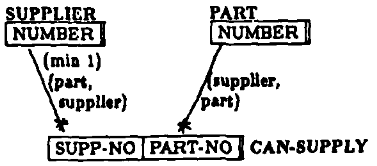


Figure 21 Partial dependency (1) of SUPPLIERS on PARTS of $M{:}N$ cardinality

straints, a no-dependency relationship of cardinality $M{:}N$ is represented by any of the three-relation representations. By restricting one or both of the connecting attributes in the connecting relation to unique values, the cardinality becomes $1{:}N$, $N{:}1$ or 1:1. Figure 20 shows examples using the association connecting relation. The connecting relation could also be a primary or a nest relation, with the same consequences for cardinality and dependency.

In Figure 20b, the cardinality of EMPLOYEE:DEPARTMENT is $N{:}1$ because of the uniqueness constraint specified on the connecting attribute EMP-NO. If an employee has a department, it can only be one department. In Figure 20c, both connecting attributes are constrained to be unique. If a manager manages a department, it is only one department, and if a department has a manager, it is only one manager.

A partial dependency (i) of cardinality $M{:}N$ is represented by attaching a (min i) on the connection from the relation that represents the dependent object class to the connecting relation. Figure 21 shows a partial dependency (1) of SUPPLIERS on PARTS. The total dependency (i,j) is represented by an additional (min j) constraint on the other connection.

A partial dependency (i) for $1{:}N$ cardinality is represented by three relations, similarly to the $M{:}N$ case. Figure 22 shows an example, a partial dependency (10) of DEPARTMENTS on EMPLOYEES. Here, every department must be related to at least 10 employees.

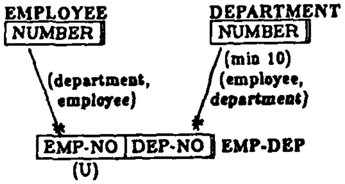


Figure 22 Partial dependency (10) of DEPARTMENTS on EMPLOYEES of $1{:}N$ cardinality

	Dependency	Cardinality of A:B
Two-relation representation	Partial B on A, total dependency	1:1, $1{:}N$
Three-relation representation	No-dependency, partial A on B, and B on A, total dependency	1:1, $1{:}N$, $M{:}N$

Table 5 Structural properties represented by two and three relations

Partial and total dependencies for 1:1 and $1{:}N$ relations can also be represented by three relations by attaching the constraints to the appropriate connections.

Table 5 shows the cardinality and dependency properties that can be represented using the two-relation representation and the three-relation representation. We assume a relationship A:B. For the two-relation representation, we assume the connecting attribute is in the main relation that represents class B.

5.3.2 Maintaining the structural properties of a relationship

In the previous section, we saw how the structural properties of a binary relationship can be represented in the structural model. We do not only want to specify the structural properties, but also how to maintain these properties when a requested transaction to the database causes a violation of a structural property. The model thus describes the behaviour of the database in the face of update transactions. As we shall see, the alternatives are not too numerous, and can be described adequately with the connections of the structural model.

In Section 5.3.2.1, we discuss how structural properties of a relationship can be maintained when objects are added and removed from the object classes that participate in the relationship. We then show in Section 5.3.2.2 how these properties are maintained in a structural model representation of the relationship.

5.3.2.1 *The insertion and deletion rules*

When objects that participate in a relationship are created and deleted, the transaction must maintain the cardinality and dependency properties of the relationship. Recall from Section 2.2 that relating properties of an object class are used to specify the relationships of this object class with other object classes. The basic constraint then is that values of a relating property must reference existing objects in the other object class. We call this the *relating property* or RP constraint.

Definition 20: The RP constraint specifies that for every relating property P of object class A that serves to define a relationship A:B, the value of P for every object in A must reference an existing object in B

Insertion of objects:

Insertion of a new object can violate the RP constraint if the value of some relating property of the object references a non-existing object. In this case, we prohibit the insertion of that object, or we insert the object, and the object it references.

For example, in Figure 1 (Chapter 2), the values for the relating properties DEPARTMENT and PROJECTS of the EMPLOYEES object class must reference existing department and project objects from the DEPARTMENTS and PROJECTS object classes, respectively.

Whenever a new employee object is created, any values for the properties DEPARTMENT and PROJECTS must be checked to ensure that they reference existing department and project objects. If the value of a relating property for some object does not satisfy this RP constraint, some error exists, and the erroneous relating property value must be checked and corrected.

It may be that a new department or project is being created. In this case, it is necessary to add the new department or project object first. In the case of total dependencies, it is sometimes necessary to add new objects into more than one object class simultaneously by an insertion transaction.

For example, consider a 1:*N* relationship DEPARTMENTS:EMPLOYEES that is also a total dependency (10). When a new department object is created, at least 10 new employee objects must be related to this new department. Hence, either new employee objects are created and inserted with the new department in a single transaction, or existing employees are related to the new department when the department is created in a single transaction, or a combination of both new and existing employees are related to the new department in a single transaction.

Insertion can also cause a violation of the cardinality and dependency constraints. If an attempt is made to insert a tuple which violates either constraint, the insertion is rejected.

Hence, for the insertion operation, any attempt to insert an object which violates the structural properties of a relationship is prohibited. If groups of objects are inserted as a transaction, the result of the complete transaction must not violate the structural properties of the relationship.

Deletion of objects:

Now consider deletion. Deletion can also cause inconsistencies to structural properties of a relationship. For example, consider a partial dependency relationship of EMPLOYEES on DEPARTMENTS. A request to delete a department object that is related to some employee will cause an inconsistency.

There are two choices to maintain the consistency of the relationship. The first is to prohibit such a deletion. We call this the *prohibit deletion* or PD rule. The second choice is to delete the specified department object, and also delete the employee objects that are related to the department object in order to maintain the dependency property of the relationship. We call this the *delete* or DLT rule.

Formally, given a relationship A:B, such the A is dependent on B. The dependency could be a (min 1) or a (min i), and we do not care whether B is dependent on A or not. Consider a deletion transaction that removes an object from B which causes the violation of the specified dependency. The following two rules can be used to prevent the inconsistency from occurring:

Definition 21: The DLT rule specifies that objects in A whose existence becomes inconsistent because of the deletion of the object from B are deleted, if their deletion is possible; otherwise the deletion of the object from class B is rejected.

Definition 22: The PD rule specifies that the deletion request for the object from class B is rejected.

The choice between these two rules depends upon the situation. If the relationship is of a hierarchical (owner object-owned objects) nature, the DLT rule should be specified. If both object classes that participate in the relationship are of equal importance, the PD rule should be specified.

If the DLT rule is specified, then we must verify that the deletion of the related objects will not cause an inconsistency. The object can only be deleted if all other objects that must be deleted with it can be deleted, without creating an inconsistency; otherwise the deletion request is rejected.

The PD rule forces an additional check when deleting objects. If a deletion is prohibited, either the deletion is erroneous, or other objects in the database—the ones that will cause the inconsistency—must either be deleted, or related to other objects before that object is deleted.

When the DLT rule is specified on a 1:*N* relationship A:B which is a partial dependency B on A, the rule represents a hierarchical owner-owned relationship. However, when the DLT rule is specified on a partial dependency (i) of A on B, the rule does not represent an owner-owned hierarchy. For example, consider a partial dependency (1) of A on B. Here, deletion of objects from A does not cause inconsistencies, while deletion of objects from B can cause an inconsistency if it is the only object related to an object of class A. In this case, the DLT rule specifies that when the last B object that is related to an A object is deleted, this A object is automatically deleted.

A total dependency relationship A:B will require two rules: one if the inconsistency is caused by deleting a class A object, and the other if the inconsistency is caused by deleting a class B object. Hence, we must specify two rules to maintain the total dependency. The two rules do not have to be the same—one rule could be DLT and the other PD .

5.3.2.2 *Maintaining relationship constraints in the structural model*

The rules for maintaining the structural properties of the representation of a relationship in the structural model are quite similar. At the model level, we talk of relations, tuples, and attributes rather than object classes, objects, and properties, respectively. The correspondence is not direct, however, since connections in the structural model do not directly represent $M{:}N$ relationships.

The RP constraint is directly specified by the connection specifications. Recall that all tuples at the N side of an ownership or a reference connection, or at the from side of an identity connection, must be connected to tuples on the other side of the connection. Hence, the connecting attributes of relations at the N side of an ownership or a reference connection, or at the from side of an identity connection, obey the *connecting attribute*, or CA constraint, which is analogous to the RP constraint. This is so because these attributes are used mainly to relate objects. We will call these attributes the *basic connecting attributes*.

Note that tuples of relations on the 1 side of an ownership or reference connection, and on the to side of an identity connection, do not have to be connected to tuples from the relation at the other side of the connection, and hence the connecting attributes in these relations do not obey,in general, the CA constraint. This is so because these attributes are used to identify the objects, as well as relate them to other objects. We will call these attributes the *auxiliary* connecting attributes.

When a (min i) is specified on a connection, the auxiliary connecting attributes obey the CA constraint, because they are now used to enforce a dependency constraint on the object class.

Two-relation representations:

Consider the two-relation representations of Figure 18. An ownership connection (A,B) and a reference connection (B,A) both represent a partial dependency of B on A. The ownership connection maintains the consistency using the *DLT* rule (delete connected tuples), and the reference connection maintains the consistency using the PD rule (prohibit deletion of connected tuples). Note that we are now dealing with tuples that represent objects, rather than objects themselves. Here, the CA constraint on the basic connecting attributes represents the RP (relating property) constraint, and since the basic connecting attributes are located in the relation that represents B, the CA constraint also represent the dependency of B on A.

The two-relation representation can also represent a total dependency if a (min i) is specified on the connection. Figure 23 shows an example, where a total dependency (1) of a $1{:}N$ relationship DEPARTMENTS:EMPLOYEES is represented by a reference connection. Here, every employee must be related to a department, and every department must be related to at least one employee. Now, the CA attribute is enforced on the auxiliary connecting attributes to represent the dependency (1) of DEPARTMENTS on EMPLOYEES.

The reference connection maintains the dependency of EMPLOYEES on DEPARTMENTS by the PD rule. It does not specify how the dependency of DEPARTMENTS on EMPLOYEES (that a department must be related to at least one employee) is maintained. If only one employee was related to a department, and an attempt is made to delete that employee, the (min 1) constraint is violated. Hence, when a connection is further constrained by a (min i), another rule for the maintenance of this constraint must be specified on the connection. This rule can be either DLT or PD , and need only be specified when a (min i) is attached to the connection. By convention, we omit the additional rule if it is the same as that specified by the connection (DLT for ownership, PD for reference).

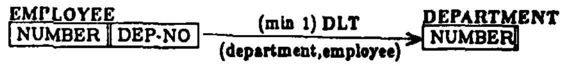


Figure 23 A connection with a (min 1) constraint

In Figure 23, the DLT rule is specified to maintain the (min 1) constraint, which means that when the last employee in a department is deleted, the department is also deleted. If a (min 10) was specified as a means to warn that a department will have less than 10 employees, the PD rule would have been specified. In the latter case, the employee will not be deleted unless either the constraint is changed (to a (min 8), say), or a new employee replaces the one being deleted.

The number of different two-relation representations is 12, 6 with cardinality 1:1 and 6 with cardinality $1{:}N$. For a given cardinality, 3 cases exist with each of the ownership and reference connections: a connection without a (min i), and two cases for the connection with the (min i) specified—one for DLT and one for PD rule.

For a relationship A:B, the two-relation representations, with the basic connecting attribute located in the relation that represents class B, can hence represent either a $1{:}N$ or 1:1 cardinality. For each cardinality case it can represent a partial dependency of B on A—when no (min i) constraint is specified—or a total dependency (i)—when a (min i) constraint is specified. An ownership connection specifies the DLT rule to maintain the dependency of B on A, while a reference connection specifies the PD rule. For total dependencies, either DLT or PD can be specified to maintain the dependency of A on B.

Three-relation representations:

Now consider the three-relation representations. Many possibilities exist, some of which may not be used very frequently. We will enumerate the possibilities after giving a few examples. There are three representations: association, nest of references, and primary. Each representation can represent a no-dependency, partial A on B, partial B on A, or total dependency of any cardinality by appropriately constraining the connections and the connecting attributes.

If neither connection is constrained by a (min i), a no-dependency relationship is represented. Tuples can be inserted in the two main relations R_a and R_b when objects that correspond to these tuples are introduced. At any time, when a relationship between two objects is recognised, a connecting tuple is inserted in R_{ab}, and when a relationship between two objects is terminated, the connecting tuple is deleted.

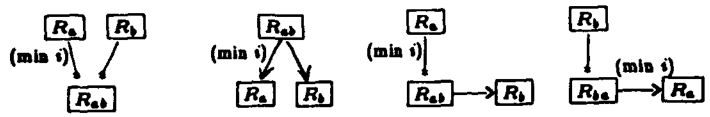


Figure 24 Partial dependency (i) of A on B represented by three relations

The association of Figure 17b(i) does not impose any deletion constraints on tuples in R_a and R_b. When a tuple is deleted, the connecting tuples are automatically deleted. Hence, we choose this representation if objects can be removed regardless of whether they are related to objects of the other class.

When only objects that are not related can be deleted, we choose the primary connecting relation, shown in Figure 17b(ii). In this case, the connecting tuple must first be explicitly deleted before a tuple in R_a or R_b can be deleted. Hence, this representation monitors the existence of tuples in R_a and R_b more closely, and requires an additional check before tuples are deleted from R_a or R_b. Only tuples that are not related can be deleted.

The nest of references of Figure 17b(iii) carries out the check on only one of the objects classes, the one connected to R_{ab} via the reference connection. The choice between these three representation hence depends on how carefully we want to check deletion of objects.

Now consider partial dependencies. To be specific, we consider a partial dependency (i) of A on B (Figure 24). Every tuple in R_a must be connected to at least i connecting tuples in R_{ab}. Hence, a new R_a tuple can only be inserted via a transaction which specifies the i tuples in R_b that will be connected to it, so that the R_a tuple and at least i tuples in R_{ab} are inserted in the transaction.

For deletion, we must specify either the DLT rule or the PD rule to maintain the (min i) constraint. In most cases, the PD rule makes more sense. If an attempt is made to delete a tuple in R_{ab} which reduces the number of tuples connected to a tuple in R_a to less than i, the constraint is violated and the deletion prohibited. Note that the deletion of tuples in R_{ab} could be initiated by deletion of a tuple from R_b for the association and the nest of references (B,A) of Figure 24. In this case, the deletion of the R_b tuple is prohibited.

If the *DLT* rule is specified to maintain the (min i) constraint, then when deletion of a tuple from R_{ab} violates the constraint, the connected tuple in R_a and all tuples connected to it in R_{ab} are deleted. This usually does not make sense except for some cases where $i = 1$.

To represent a total dependency, a (min i) is attached to one connection and a (min j) to the other connection. A PD or DLT rule must be specified on each connection to give the desired rule for consistency maintenance. In most cases, the PD rule is more plausible, as in the partial dependency case.

For insertion, a transaction to insert a tuple in R_a must also relate that tuple to at least j tuples in R_b. Hence, the transaction should also insert at least j tuples in R_{ab}.

When a (max N) is attached to a connection, it only constrains insertion in the obvious way. Insertions are also constrained to obey the cardinality constraints. Cardinality and (max N) constraints do not constrain deletions.

For a three-relation representation of a relationship, there are three main cases, depending on the pair of connections used (or on the type of connecting relation). These are the association, primary and nest of references (Figure 17). For each of these representations, we could have a 1:1, 1:N, or M:N cardinality, and for each cardinality we have nine cases for the constraining of connections for the non-symmetric case (nest of references), and six cases for each of the symmetric representations (association and primary). Hence, we can represent 27 distinct cases using nest of references (3 × 9), and eighteen distinct cases (3 × 6) for each of the association and the primary representations. The reason for the different number of cases is that the nest of references has two connections of different types, while the association and primary representations have two connections of the same type.

For an association or a primary representation of a given cardinality, one case represents a no-dependency relationship (when no min constraint is specified on either connection), two cases represent a partial dependency (when the connection with the min constraint specifies the DLT or PD rule), and three cases represent a total dependency (when the rules on the min are either two DLT 's, two PD 's or one DLT and one PD).

For a nest of references of a given cardinality, one case represents a no-dependency relationship (when no min constraint is specified on either connection), four cases represent a partial dependency (two cases when the min constraint is specified on the ownership connection, and two cases where the min constraint is specified on the reference connection), and four cases represent a total dependency (combining the specification of DLT and PD on the two connections).

Hence, we have a total of sixty-three distinct cases. Twenty-one cases exist for each cardinality. For a given cardinality, three cases represent a no-dependency, eight cases represent a partial dependency, and ten cases represent a total dependency.

The reason for the large number of cases is that we are dealing with three relations. In our classifications for a relationship between two object classes, we were only dealing with two object classes. Hence, in a sense, the structural model is too fine in its representation of relationships.

5.4 REPRESENTING OTHER SEMANTIC CONCEPTS

In this section, we discuss the representation of the remaining semantic concepts of Chapter 2. These concepts include subclasses (Section 2.1.3), abstraction classes (Section 2.1.2.2), and aggregate classes (Section 2.1.2.3). We also show how to represent a relationship between categories of object classes, as discussed in Section 2.2.3.

In Section 5.4.1, we discuss the representation of subclasses. We show how subclass constraints are maintained, and how users can access the properties of objects that belong to a subclass. In Section 5.4.2, we examine the representation of abstraction and aggregation classes. We show that abstraction and aggregation classes are just special types of relationships. Finally, in Section 5.4.3, we show how to represent relationships between categories.

5.4.1 Representing subclasses

As discussed in Section 2.1.3, every property of an object class implicitly specifies a set of subclasses for the class. Each subclass in the set contains the objects that have the same value for that property among all objects in the class. In the structural model representation, the tuples of the main relation that represents the object class are similarily implicitly divided into sets of tuples by the values of each attribute.

These implicit subclasses do not have to be expiicitly represented in a data model. The tuples in a relation that belong to such a subclass can be accessed for retrieval by specifying the appropriate logical expression on the relation in an SMQL query. For example, the subset of employee tuples with job value engineer can be specified by the following SMQL query:

Q31: GET EMPLOYEE
WHERE JOB of EMPLOYEE = *Engineer*

Furthermore, subsets of tuples defined by a set of values for an attribute can similarily be specified. For example, the subset of technical employees can be specified by:

Q32: GET EMPLOYEE
WHERE JOB of EMPLOYEE ∈ {*Engineer, Researcher, Technician*}

If such a subset of tuples is referred to frequently by the users, it is useful to refer to the subrelation by a name. A derived subrelation with the appropriate name can be created.

DEFINE DERIVED RELATION TECHNICAL-EMPLOYEE TO BE GET EMPLOYEE
WHERE JOB of EMPLOYEE ∈ {*Engineer, Researcher, Techsician*}

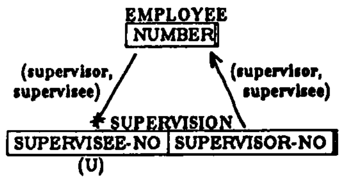


Figure 25 The SUPERVISION relationship

For the vast majority of cases where reference to a subset of tuples—which represents a subclass of objects—is desired by the user, the definition of such derived subclasses will be all that is needed. The definition of these subclasses is a means of quick reference by name to the subclass of objects.

Many relationships can be used to define such subclasses. For example, consider the SUPERVISION relationship, and assume that it is a no-dependency, 1:*N* relationship, so that an employee can have at most one supervisor. It can be represented as shown in Figure 25 by a nest of references from EMPLOYEE to EMPLOYEE.

We can define the object class SUPERVISORS by a derived relation SUPERVISOR in the SMQL as:

DEFINE DERIVED RELATION SUPERVISOR TO BE
GET EMPLOYEE WHERE COUNT supervisee of EMPLOYEE ≥ 1

All tuples in EMPLOYEE that are referenced by at least 1 tuple from the SUPERVISION relation via the reference connection (supervisor,supervisee) are hence in the derived subrelation SUPERVISOR. In Figure 25, we chose to constrain deletion of employees that are supervisors. However, it is equally valid to replace the reference connection with an ownership connection and remove this constraint. This depends on the user requirements.

In this example, any employee can become a supervisor simply by insertion of a tuple in the SUPERVISION relation that relates him as a supervisor of an employee. If more control is required, we can represent an explicit subclass of employees called supervisors. An employee can be a supervisor now only if explicitly inserted in the SUPERVISOR subrelation (Figure 26). Hence, more control is exercised than in the previous representation.

Such an explicit subclass is represented in the structural model by a subrelation connected to its base relation via an identity reference connection. Explicit subclasses are only represented when attributes or connections or both are defined that apply to the tuples in the subrelation only, or when the tuples in the subclass have to be explicitly specified independent of attribute values of the connected tuple in the base relation.

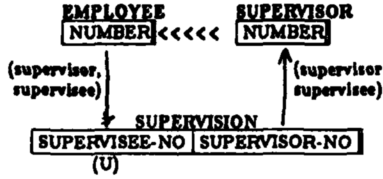


Figure 26 Explicit representation of the SUPERVISOR relation

Such explicit subclasses are represented in a data model in any of the following three cases:

(1) To remove the need for optional attributes that are only applicable to some tuples in a relation. Such tuples can belong to an explicit subrelation, and the attributes that apply only to these tuples that are members of the subrelation are defined as attributes of the subrelation.

(2) As a means of explicit integrity control of the representation or when constraints specified by connections only apply to a known subset of the tuples in a relation. The example of Figure 26 illustrates this point. The SUPERVISION relationship is between the classes EMPLOYEES and SUPERVISORS, and SUPERVISORS is a known subclass of EMPLOYEES. Hence, the representation of the relationship restricts it to a relationship between EMPLOYEES and the subset of EMPLOYEES that have been explicitly declared to be supervisors.

(3) When a subclass of objects from a given class has a semantic significance, but that subclass cannot be created by conditions on the values of attributes in the tuples. An explicit non-restriction subrelation is created in this case, and only tuples that are explicitly inserted in that subrelation by the users belong to the subclass.

The inheritance rule for a subrelation allows all attributes in its base relation to be referenced automatically by the SMQL (see Section 4.3.7).

For example, all attributes in the EMPLOYEE relation (none are shown in Figure 26, but see the attributes in Figure 15 for EMPLOYEE) can be accessed directly from the SUPERVISOR relation. The query "What is the name of the supervisor of *John Smith*" can be written as:

Q33: GET (FIRST-NAME, LAST-NAME) of supervisor of EMPLOYEE
WHERE FIRST-NAME = *John* AND LAST-NAME = *Smith*

Note how the attributes FIRST-NAME and LAST-NAME of EMPLOYEE are inherited by the subrelation.

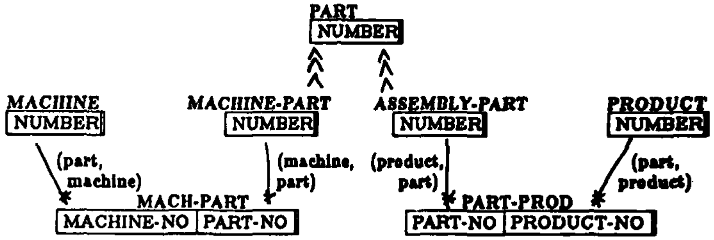


Figure 27 The PART relation and its subrelations

Additional attributes that are of the subrelation only cannot be accessed automatically in the SMQL via the name of the base relation. The reason for this restriction is twofold. First, the attributes in the subrelation apply only to tuples in the subrelation, and so most references to the attribute will be by referring to the name of the subrelation. Second, we did not want to overcomplicate the connection tracing mechanism of the SMQL (see Section 4.3.7).

For example, if the SUPERVISOR subrelation in Figure 26 had an additional attribute NO-OF-YEARS that shows the number of years of being a supervisor, we cannot access it as an attribute of EMPLOYEE. The reason is that the additional attributes apply only to employees who are supervisors, and hence should be referenced only when specifying a supervisor. Hence, we can only reference NO-OF-YEARS as an attribute of SUPERVISOR, but not as an attribute of EMPLOYEE.

Sometimes, it is convenient to refer to attributes of a subrelation through the name of the base relation. In this case, either the identity connection is given a name to allow specification of the path, or an auxiliary connection is created to connect the subrelation directly to some relevant relation.

For example, referring to Figure 27, suppose the relation PART that represents the parts used in a plant of a company had two subrelations MACHINE-PARTS (parts used for machine maintenance) and ASSEMBLY-PARTS (parts used for assembling products). All parts share common identification and description properties (name, part number, color,...etc.) and the same relationship with the SUPPLIERS object class. Machine parts have additional relationships with the MACHINES object class, while ASSEMBLY-PARTS are related to PRODUCTS.

Note that the two subrelations MACHINE-PART and ASSEMBLY-PART may not even be disjoint. If we require them to be disjoint, we must include an attribute TYPE in the relation PART, whose domain is {*machine*, *assembly*}, and define the two subrelations to be restriction subrelations on the TYPE attribute.

Suppose a user desires to refer to attributes or connections of a subrelation using the name of the base relation. We must define these attributes or connections as derived attributes or connections of the base relation. For example, to refer to product of PART rather than product of ASSEMBLY-PART in Figure 27, we define the derived connection (product) from PART to PART-PROD as follows:

DEFINE DERIVED CONNECTION (product) TYPE OWNERSHIP
FROM PART (NUMBER) TO PART-PROD (PART-NO)

Note that we only need a single name for the derived connection, since we will use it only in the direction from PART to PART-PROD.

An identity connection does not require a name for SMQL processing. Whenever an SMQL statement references an attribute of a subrelation, and the attribute is not found in the subrelation, a search for the attribute is conducted in the base relation of the subrelation. If it is not found there, and the base relation is a subrelation of another subrelation, a further search is conducted, and so on.

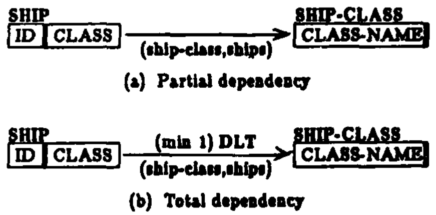
The question of ambiguity could arise if a subrelation has more than one base relation, and both have attributes of the same name. We do not allow this in the structural model. The schema processor will automatically check for such an ambiguity, which is a straightforward procedure. In most cases, a subrelation will not have more than one base relation anyway, and this ambiguity does not arise. Section 4.3.7 discusses this and other rules to enforce on the data model names to guarantee non-ambiguity of SMQL statements.

5.4.2 Representing abstraction and aggregation classes

Recall from Section 2.1.2 that an abstraction class is a class of objects that group together objects from another class with identical common values for a set of properties. This phenomenon is frequently encountered in database modelling. The properties of the abstraction class are often considered to be properties of the other class, but we have to represent the constraint that all objects of the other class that are related to the same object of the abstraction class must have the same values for all the properties of the abstraction class.

For example, an abstraction class SHIP-CLASSES groups together ship objects that have been built to the same specification, or a CAR-TYPE abstraction class groups together car objects that have been built to the same specification. Here, a group of ship or car objects share property values that depend on the class of the ship, or the type of the car.

A relationship exists between the abstraction objects and the objects upon which the abstraction is specified. The cardinality is 1:*N*, and it can be represented either as a total dependency (1) DLT or as a partial dependency of the object class on the abstraction class. This is because we do not want an object to exist that does not reference an existing object of the abstraction class.



(a) Partial dependency

(b) Total dependency

Figure 28 Representing the abstraction class SHIP-CLASSES of the object class SHIPS

Referring to the SHIP-CLASSES example, both representations prohibit deletion of a ship-class tuple as long as it is connected to any ship tuples. The partial dependency allows the existence of ship-class tuples that are not connected to any ship tuples, while the total dependency (1) causes automatic deletion of a ship-class tuple when the last ship tuple connected to it is deleted. Figure 28 shows the SHIPS:SHIP-CLASSES abstraction relationship as represented in the structural model.

It is advantageous to define the dependent attributes of the relation that represents the abstraction class as derived attributes of the relation that represents the object class upon which the abstraction is defined. This is because we usually associate the properties of the abstraction class with the class of objects itself. For example, referring to Figure 3 (Chapter 2), the properties length, draft, dead-weight, gross-weight and ship-type that will be attributes of the SHIP-CLASS relation of Figure 28 should also be defined as derived attributes of the SHIP relation. This is because one naturally associates these properties with a ship object, but the consistency constraints of the model (that all ships of the same class have the same values for these properties) require the representation as two relations.

An aggregation class (see section 2.1.2) is a class of objects, each of which consists of a set of objects of another class. The example is a class of CONVOYS, each of which consists of several ships from the SHIPS object class. This kind of relationship also occurs frequently. Another example is a DEPARTMENTS object class, where each department object consists of a set of EMPLOYEE objects.

An aggregation relationship is represented as a nest of references in the structural model. Figure 29 shows the CONVOYS:SHIPS relationship. The CONVOY relation represents the CONVOYS object class, and the nest relation SHIPS-IN-CONVOY represents the relationship. If the relationship is of cardinality 1:*N*, as in this example, the connecting attribute SHIP-ID is restricted to unique values. Here, we are assuming that the current convoys are the ones represented, so no ship can be in more than one convoy at the same time.

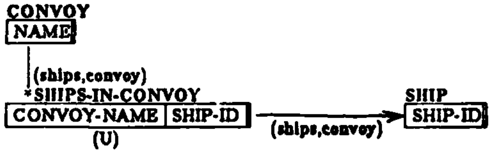


Figure 29 Representing an aggregation relationship CONVOYS:SHIPS

Hence, we see that abstractions and aggregations are just types of relationships with known structural properties, which can be represented by connections in the structural model. If they occur frequently, it is useful to analyse their structural properties, and be aware of the representation choices.

5.4.3 Representing relationships between categories

We now show how to represent a relationship between categories of object classes. Recall from Section 2.2.3 that a relationship need not only be between two object classes, but could also be between categories of object classes. A *category* is a group of object classes, and objects in the category include all objects from all the object classes in the category.

To represent a category, we define a relation, the *category* relation, that represents all objects in the category. Since the objects in the category can be from distinct classes, they may not all have the same properties. The main relations that represent the distinct object classes must hence remain distinct, each with its own attributes. What we do is augment each main relation with an *invisible* attribute—an attribute whose value cannot be referenced by the user. This invisible attribute is used to map the objects of the different classes into a uniform representation as members of the category.

Suppose a category is composed of *n* object classes. The invisible attribute is a means of mapping objects from those *n* distinct classes into a single category. This single category can then participate in the relationship as though it were a single object class. Connections from each of the object classes to this category via the invisible attributes establish the correspondence of objects from each class to their representation in the category. Note that each tuple in the category corresponds to exactly one object from exactly one of the *n* object classes.

Consider the example of Section 2.2.3, where we have an ASSIGNMENT relationship in a company between the company owned vehicles, and the company entities to which each vehicle is assigned. Let us assume that in a particular company, vehicles can be cars, trucks, or airplanes, and that a vehicle can be assigned to a department, an individual employee, or a customer. Hence, the relationship is between the object class of VEHICLES, which has the subclasses CARS, TRUCKS, and AIRPLANES, and the category of VEHICLE-OWNERS, which consists of the object classes DEPARTMENTS, EMPLOYEES, and CUSTOMERS. Note that VEHICLES is considered an object class because we assumed that some properties are common to all vehicles, such as the properties REGISTRATION-NUMBER, COLOR, PURCHASE-DATE, ...etc, and the identifying property REGISTRATION-NUMBER is among the common attributes. If this was not the case, we would have to create a category for VEHICLES also.

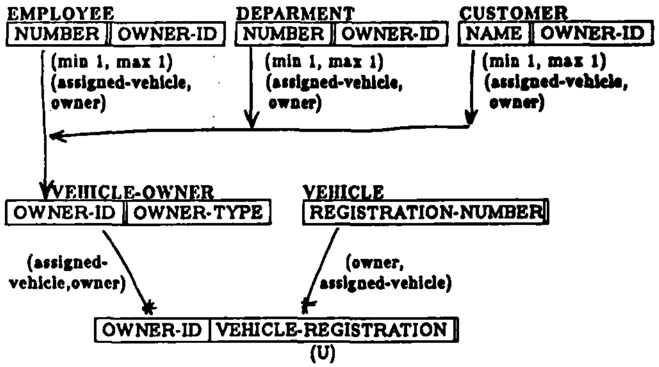


Figure 30 Relationship between a category and an object class

Figure 30 shows the representation of this ASSIGNMENT relationship in the structural model. The VEHICLE relation is the main relation that represents the VEHICLES object class, and the VEHICLE-OWNER relation is the main relation that represents the category of VEHICLE-OWNERS. The invisible attribute OWNER-ID of the relation VEHICLE-OWNER is used by each of the relations that represent the object classes of the category—EMPLOYEE, DEPARTMENT, and CUSTOMER—to reference VEHICLE-OWNER. The three reference connections from each of these relations to the category relation have the same name, in violation of our naming rules. No ambiguity actually arises, however, since each of the reference connections references a distinct subset of the tuples in VEHICLE-OWNER—the subset that represents objects of a particular class—and these subsets are disjoint. Hence, for connections defined on invisible attributes, we will allow more than one connection to have the same name in those names of connections that are in the direction from the relation with the invisible attribute. This set of connections is considered as a *single* connection in the connection name disambiguation processes of the schema (see Section 4.3.7).

A statement in the SMQL that refers to owner of VEHICLE will be traced in the schema to any of the three object classes of the category of vehicle-owners, depending upon the value of the OWNER-TYPE attribute of VEHICLE-OWNER. The domain of the OWNER-TYPE attribute is {*Employee, Customer, Department*}, and the appropriate owner connection from the category relation VEHICLE-OWNER is traced depending on the value of the OWNER-TYPE attribute in a particular tuple. It is unfortunate to include this dependency on data values in the schema, but there does not seem to be a way around it.

5.5 CHOICE OF NAMES FOR ATTRIBUTES, RELATIONS, AND CONNECTIONS

In this section, we discuss the choice of names for relations, connections and attributes for a structural model. So far, we have seen how the structural model can represent the semantic concepts of Chapter 2, so that their behavioural properties are taken into account. The structural model representations however do not always correspond directly to the concepts that a user would use comfortably to interact with the database. This is not important; the important consideration is that the language that is used for interaction with the database be close to the concepts of the user.

The SMQL is the interaction language for the structural model (see Section 4.3). In order to make the SMQL retrieval and update statements match the concepts of the user, we must carefully choose the names of attributes, relations, and connections, since these are the main constructs of the model that are used by the SMQL. If the names are well chosen, the model will be simple and easy to use.

The data model designer should have some convention for choice of relation names. The name of a relation that is the main relation representing an object class should be some name the user associates with the object class. In all our examples, we used the convention that the name of the relation is the singular form of the noun that is the name of the object class. Another possibility is to use the plural form of the noun.

The names of connecting relations are usually not referenced during retrieval, and hence can be any combination (possibly abbreviated) of the names of the object classes that the connecting relation is relating.

It may be useful to give a relation two names, one for use by casual users, and the other for use by programmers. Programmers will use the name frequently, and hence should not have to use long names. The only restriction on relation names is that they all be distinct, including duplicate names, so that no ambiguity arises.

Attribute names should correspond to the names the user associates with the property the attribute represents. We do not have to choose qualified attribute names since names of attributes do not have to be unique over the whole data model. The restriction here is that attribute names within a relation are unique. This flexibility is the result of separation of attribute names from domain names.

Again, we may choose a second abbreviated name for an attribute subject to the constraint that all attribute names within a relation are unique.

Strict adherence to the above naming conventions is very helpful when integrating data models. If different users use the same name for a main relation that represents an object class, we can easily recognise relations in different data models that represent the same object class. We discuss integration in Chapter 6.

For connection names, we choose the name that the user associates with accessing objects from one object class that are related to an object of the other class. For example, since one refers to the department of an employee, the connection name in the direction *from* an EMPLOYEE relation to a DEPARTMENT relation can be department. In the reverse direction, the connection name can be employees.

A point to consider is that when *more than one relationship exists between* the same two object classes, neither of the connection names that correspond to these relationships should be the name of the other object class, since this may cause some users to use that name while referring to the other relationship. It is better to make the names of both connection paths closer to the meaning of the relationship.

Note that whether two or three relations are used to represent the relationship, the *reference* to objects *from* the related object class is the same. In the three-relation representation (Figure 20(b)), two connections are traversed, while in the two-relation representation (Figure 18) only one connection is traversed. The form of a query that references related objects is the same in both cases, since the connection name is used only once.

It is useful to use the plural noun for a relationship when a set of objects is related to a single object of the other class, (as in employees related to a department) and the singular noun when only a single object is related (as in department of an employee). This makes the query reflect the cardinality of the relationship.

The condition for no ambiguity here is that all connection names in the direction from a relation be unique. This is easily enforced by the schema processor.

Names for derived attributes, relations, and connections should also follow the guidelines discussed, and are subject to the same disambiguating constraints.

5.6 SYSTEMATIC DATA MODEL DESIGN

In this section, we discuss the steps taken by an expected user group of the database to design a *data model* suitable for their needs. The emphasis is on a data model that is:

(1) *Correct*: represents the object classes and relationships of the situation of interest to the user group correctly and consistently, and

(2) Easy to use: both by casual members of the group, and by members of the group that are expected to make heavy, possibly routine, use of the database.

This design is carried out with the help of a database design expert. However, the definition of a user's requirements need not be technical in nature, but is rather best done in terms of real-world constructs the user is familiar with such as object classes, relationships, and rules that the user expects to hold. The expert can then design the formal data model and the constraints, and choose the names of relations, connections and attributes, in consultation with the user, so that the user will see a model very close to his or her expectations.

Information concerning expected usage of the database is important to include, even at this early stage. This type of information includes how often retrieval and update operations are performed, what operations are critical, what constraints have to be always enforced, ... etc. We do not consider this aspect in our present work. In this section, we concentrate on the design of a structural data model. This aspect of database design is often called *logical* data model design.

In this section, we discuss the design of a single data model, by one group of *expected users* of the database. In general, there will be many such groups of users. From the requirements of each group, one data model is produced. To complete the design of the database, we must *integrate* all the data models to form an integrated *database model* which correctly supports the data models of the different users. We discuss integration in detail in Chapter 6.

To make our discussion clear, we will use an example of designing a data model to be used by some international authority to monitor the movement of merchant ships around the globe. This monitoring is intended to enforce international shipping regulations, and to be used in case of an *emergency* at sea where some ship needs aid. This example is unusually complex for a single data model, but we use it to illustrate the different design steps. In general, the data model of a single user group will be far simpler than our example.

The design of a data model is an involved process, and cannot be done by one person in a short time. Rather, each subgroup of the user group identifies its information and processing requirements. Several iterations of discussions among the subgroups of the user group are undertaken, until a clearer understanding of the requirements of the whole group emerges. A *preliminary* data model is then designed, and examined by the subgroups to see that it satisfies their requirements. The database design expert should be involved from the start, and it is *very useful* to document the different design iterations so that the reasons for decisions taken are clear.

Figure 32 shows the steps taken to design a data model. The steps are not strictly ordered, and iterations can be carried out to create *revised* versions of the model at the different steps.

STEP	DESCRIPTION
1	Identify object classes of the situation
2	List the properties for each object class
3	Examine each object class for subclasses with different properties
4	Examine each object class for properties that share common values for groups of objects in the class
5	Identify ruling properties for each object class, and choose one set to be the ruling part
6	Identify the relationships between object classes
7	Design minimal relations of the basic data model
8	Design minimal connections of the basic data model
9	Augment the basic model by defining all derived attributes, relations and connections

Figure 31 Relationships between SUPPLIERS, PARTS, and PROJECTS

The structures that finally appear in the data model can be separated into two parts. The first part, which we call the *basic* data model includes all structural requirements and domain constraints that are expected to hold in the database. This basic model should contain a minimum of redundant representations so that as few additional constraints as possible are needed to ensure the correctness of the representation. Redundant representations require the specification of constraints to ensure that they *represent the same things.*

Hence, the basic data model is a structural data model with minimum redundancy, no derived attributes, and all structural connections that impose constraints on the model defined. Additionally, important non-structural (domain) constraints that must hold in the database at all times are identified, and attached to the data model.

The basic data model might not include all constructs to which the user wants to refer to. The final data model, which we call the *augmented* data model, involves augmenting the basic data model with any additional user requirements that have been left out of the basic data model because of the minimum redundancy requirement. It involves augmenting the basic model with name synonyms, and derived attributes, relations, and connections so that the model will include all the constructs required by

the users, and be convenient to use by all members of the group. Note that we are designing a single data model for one user group, so we do not address the problem of multiple views here. Once several users of the database have defined their data models, we integrate the models as we discuss in Chapter 6.

During the design phase, both basic and augmented structures are designed. However, it is very important to separate the basic model from the augmented structures, since the basic model includes the constraints to be imposed on the database. Hence, the basic model is the one mainly used in data model integration (discussed in Chapter 6), and in considering the constraints that will be implemented on the physical database. The augmented model is the basic model plus the additional names and derived structures that facilitate the use of the basic model, and is used when choosing the physical storage organisation of the data elements, and for the design of the access paths to meet the expected usage requirements of the database.

We now give the steps followed to design the data model. We illustrate each step with the design of the ships monitoring data model. We discuss each step in separate section.

5.6.1 *Identify object classes of the real-world situation*

We identify each object class, and give it a name, and a short informal description about its meaning, and what kind of objects are in the class. It is very useful to list some objects of each class to make it absolutely clear what each class represents. This also helps to identify ruling attributes, since we usually refer to objects of the class using an identifying attribute.

Here, the user identifies the object classes that he or she want to refer to as object classes. Some of the object classes may turn out to be derived from others, but we do not worry about this now.

The output of this step is a set of object classes $\{O_1, O_2, \ldots, O_n\}$, and with each object class is associated a name, a description, and descriptions of some of the objects in the class. The set should include as many object classes as the user wishes to refer to, without consideration of redundancy. The set of object classes listed in this step need not be complete, since other classes are discovered during the design process.

SHIPS MONITORING DATA MODEL:

Throughout the design steps, we will follow the approach of a user group. We assume that the data required by the group is the data given here. A different user group may need different data, which may lead to the design of a different model. If the user group does not have some expectation of its data requirements, then clearly it should not participate in the data model design process, but could still use the database based on some other user data model if appropriate.

Figure 32 shows the preliminary object classes identified by our expected user group.

object class name	*description*	*some objects of the class (optional)*
SHIPS	merchant ships	
SHIP-OWNERS	companies that own ships	
SHIP-CHARTERERS	companies that charter ships	
SHIP-TYPES	type of ship	tanker, trawler, fishing
CARGO-TYPES	individual cargoes	crude oil, gasoline, cotton, wheat, cars
CARGO-CLASSES	cargo classes	oil, grain, machinery
PORTS	cities with port facilities	Oakland, Alexandria, Marseilles
COUNTRIES	coutries of the world	U.S.A, Egypt, France, Paraguay
VOYAGES	information about a ship voyage	

Figure 32 Initial object class identification

5.6.2 For each object class, list its properties

We list the properties of each object class defined thus far. At this stage, we do not separate types of properties (for example relating properties, optional properties, compound properties, ...etc.). This is just a preliminary identification to define a starting model. The information at this stage is kept, but successive refinement will lead to the final model. Again, we do not worry about redundancy at this stage.

We will distinguish static properties—properties that do not change frequently—from dynamic properties whose values undergo frequent update. This is a natural distinction, and is also useful in designing the physical database later on. Also, usually static and dynamic properties are used in different types of transactions on the database.

Formally, for each object class O_i identified in the previous step, a list of property names $\{p_{i_1}, p_{i_2}, \ldots\}$ is identified.

In this step, we may find some object classes that are not completely defined, and we have to re-examine the real-world situation to see what the classes mean exactly. Also, in the process of listing properties, we may discover some properties that do not fit well within any of our object classes, and have to define additional object classes. Examples are given in the model we are designing.

SHIPS MONITORING DATA MODEL:

In Figure 33(a), we show the desired properties of the object classes shown in Figure 32.

The properties shown are all the properties the users think of, and expect to reference, as properties that describe an object of the class.

By a closer examination, we find that the VOYAGES object class is not sufficiently defined. In particular, is a voyage specified between only two ports, or can it be divided into several legs, each leg being the part of the journey between successive ports. We will assume here that a voyage is the more general notion, being a journey that typically originatcs or terminates at a home port. In this model, we will not constrain a voyage to originate or terminate at a home port, since exceptions to that rule are common.

object class name	*properties of the object class*
SHIPS	static: name, hull-number, ship-class (includes all ships built to the same specification), draft, length, beam (width), dead-weight, gross-weight, cargo-capacity, fuel-type, fuel-capacity, cruising-speed, fuel-consumption (at cruising) maximum-speed, owner, country-of-ownership, country-of-registry, ship-type, possible-cargo-classes, crew size, ircs (international radio call signal)
	dynamic: charterer, last-reported-position, estimated-current-position, departure-port, destination-port, cargoes-on-board
SHIP-OWNERS	static: name, nationality, address
SHIP-CHARTERERS	static: name, nationality, address
SHIP-TYPES	static: name, possible-cargo-classes
	dynamic: number-of-ships (of this type)
CARGO-TYPES	static: name, quantity-unit, cargo-class (to which it belongs)
CARGO-CLASS	static: name, ship-types (that can carry cargoes of this type), cargo-types, number-of-cargo-types (in this class)
PORTS	static: name, country, number-of-docks, warehouses, docks, loading-capabilities, unloading-capabilities, maximum-draft, maximum-length, maximum-beam
	dynamic: ships-in-port, cargoes-in-port
COUNTRIES	static: name
VOYAGES	static: ship, departure-port, departure-time, destination-port, arrival-time, cargo-consignment, charterer

(a) Properties of the initial object classes that are defined in Figure 32

VOYAGES	static: ship, departure-port, departure-date, destination-port, arrival-date, charterer
LEGS	static: ship, voyage, leg-number, departure-port, departure-date, destination-port, arrival-date, cargo-consignment, track-positions (of leg)
STOPS	static: ship, voyage, stop-number, port-name, arrival-date, departure-date, cargoes-unloaded, cargoes-loaded

(b) Decomposition of VOYAGES into LEGS and STOPS

Figure 33 Properties of the object classes for the SHIPS MONITORING data model

A voyage can hence be broken down into several legs, and can also be broken down into several stops at ports. Legs and stops can be considered as "duals", since each leg joins two stops and each stop (except the first and the last) joins two legs. Hence, in the basic model, it is sufficient to represent either the legs or the stops of the voyage. The augmented model can then include the other concept derived from the basic model. Here, we define both legs and stops of a voyage, since both may be used by the data model users.

Note that legs and stops can be considered as separate object classes, or as repeating properties of a voyage. One of the strong points of the structural model is that the representation will be the same whether they are considered separate object classes, or properties of the VOYAGE object class. In many cases, it is difficult to decide whether such constructs are object classes or properties. Here, we consider them object classes.

The TRACK-POSITIONS property of LEGS is a compound repeating property composed of the simple properties LONGITUDE, LATITUDE, DATE, TIME, and REPORTER (of position).

In some cases, many properties exist, and then we have to decide how fine a representation of the situation we need. Some properties and object classes that clearly exist in the situation may not be of much use to the user group, and hence a coarser representation can be designed by omitting some properties and object classes. Another reason for ommitting some properties is that data values for these properties cannot be obtained, or are very difficult and expensive to obtain.

For example, a more detailed description of the PORTS object class would include detailed information on the docks, warehouses, and loading-unloading equipment at the port. In this case, we include a compound repeating property DOCKS composed of the simple properties DOCK-NUMBER, MAX-LENGTH, MAX-BEAM, MAX-DRAFT, and SHIP-IN-DOCK (the ship at the dock, if any). In Figure 33(a) we only show the name of the compound repeating property DOCKS. Similarily, a compound repeating property WAREHOUSES is included. This property is composed of the simple properties WAREHOUSE-NUMBER, POSSIBLE-CARGO-CLASSES, CAPACITY, and *CURRENT-CARGOES-STORED*, where *CURRENT-CARGOES-STORED* is a compound repeating property composed of CARGO and QUANTITY (we assume several different cargoes can be stored in a warehouse). Again, these properties are not shown in Figure 33.

The choice between the detailed and the rough description depends upon the users' needs and the availability of the data. We will use the more detailed description for the rest of our example.

5.6.3 Examine the object classes for subclasses with different properties

Some of the object classes defined will have subclasses of objects that differ from other objects in the class. We examine all object classes for such subclasses, since they will be explicitly represented in the structural data model. Some of the subclasses may have been already noted in previous steps of the design.

Formally, for each object class O_i, some subclasses $\{O_{i_1}, O_{i_2}, \ldots\}$ may be identified at this step.

In many cases, event object classes will posses different types of objects: those that describe past, present, and future occurrences of the object class. Each of these subclasses may have different properties, since more information is known about past events. In our example, three event object classes exist: VOYAGES, LEGS, and STOPS.

SHIPS MONITORING DATA MODEL:

Three subclasses of the VOYAGES object class can be distinguished—past, present, and future voyages. Future voyages are represented only if data about future itineraries of ships is available. This is particularly useful if the database is to be used in planning. However, we will assume here that information about future itineraries of ships is not available, nor is it needed, since this database is for monitoring ship movements and not for planning. Hence, we only consider past and present voyages.

object class name	*properties of the object class*
PAST-VOYAGES	ship, departure-port, departure-date, destination-port, arrival-date, charterer
CURRENT-VOYAGES	ship, departure-port, departure-date, destination-port, expected-arrival-date, charterer
PAST-LEGS	ship, voyage, leg-number, departure-port, departure-date, destination-port, arrival-date, cargoes-on-board, (reported) track-positions *(of leg)*
PRESENT-LEGS	ship, (current) voyage, leg-number, departure-port, departure-date, destination-port, (estimated) arrival-date, cargoes-on-board, (reported) track-positions (of leg)
FUTURE-LEGS	ship, (current) voyage, leg-number, (estimated) departure-port, (estimated) departure-date, destination-port, (estimated) arrival-date
PAST-STOPS	ship, voyage, stop-number, port-name, arrival-date, departure-date, cargoes-unloaded, cargoes-loaded
PRESENT-STOPS	ship, (current) voyage, stop-number, arrival-date, (estimated) departure-date, cargoes-unloaded, cargoes-loaded
FUTURE-STOPS	ship, (current) voyage, stop-number, (estimated) arrival-date, (estimated) departure-date

Figure 34 Subclasses of VOYAGES, LEGS and STOPS

Past completed voyages already have values for all the properties previously shown. The present voyage for each ship does not have all the destination data. If information is being kept on the current voyage of a ship, and updated while the ship is at sea by reports, then for each current voyage some legs will have already been traversed (and hence become past legs), one leg will be the current leg of the ship, and some legs will be projected to happen (future legs). For past legs, all information is available. For the present leg, destination information is estimated, while for the future legs, both arrival and departure information is estimated.

Similar distinctions can be made for the STOPS object class. The subclasses for VOYAGES, LEGS, and STOPS are shown in Figure 34.

5.6.4 *Examine the properties of each object class for sets of properties that share common values for groups of objects in the class*

This step corresponds to identifying abstraction classes that might have been overlooked in the initial design of the data model. It also corresponds to the normalisation step in a relational model where a set of attributes that are functionally dependent on a non-key attribute in a relation are removed into a separate relation. In some cases, the abstraction class will have already been defined as a separate object class, so this step does not apply to all abstraction classes. In our example, CARGO CLASSES is one such class that has already been defined.

We are now separating properties that are part of the basic model from those that are part of the augmented model. It is often the case that the properties of the abstraction class are directly referenced by users as properties of the object class that references the abstraction class. All such properties will be defined as derived properties of this object class in the augmented data model, as discussed in Section 5.6.9, but are only properties of the abstraction class in the basic data model.

Formally, for each object class O_i with the properties $\{p_{i_1}, p_{i_2}, \ldots\}$, each subset of properties that are constrained to always have the same values for a group of objects in the class is identified, and a new object class O_{i_j} that will be part of the basic data model is created.

This step and the previous step are not necessarily carried out in a particular order.

SHIPS MONITORING DATA MODEL:

Only one case of such an abstraction object class that has not been defined is discovered in our model, the object class that groups together ships according to their class. Ships that belong to the same ship class are ships built to the same specification. Hence, they share the values for the properties SIIP-CLASS, DRAFT, LENGTH, BEAM, SHIP-TYPE, CARGO-CLASSES, CARGO-CAPACITY, and CREW-SIZE. Here, we assume that if one of the ships of a ship class is modified in a major way, which results in a change to the value of any of the above attributes for that ship, the ship will belong to another (new) ship class.

Hence, the SIIPS object class is decomposed into two basic object classes—SHIPS and SHIP-CLASSES—as shown in Figure 35.

object class name	*properties of the object class*
SHIPS	name, hull-number, owner, country-of-ownership, country-of-registry, ircs,last-reported-position, estimated-current-position, departure-port, destination-port, cargoes-on-board, charterer
SHIP-CLASSES	ship-class-name, draft, length, beam, dead-weight, gross-weight, cargo-capacity, fuel-type, fuel-capacity, cruising-speed, maximium-speed, fuel-consumption, ship-type, possible-cargo-classes, crew-size, number-of-ships (in class)

Figure 35 The abstraction class SHIP-CLASSES of the SHIPS object class

5.6.5 Identify ruling properties of each object class, and choose one set of ruling properties for use as ruling part in subsequent design stages

This stage is straightforward. All sets of identifying properties are determined, and a single set is chosen as ruling part. The other sets of identifying properties will be made into lexicon relations, or will be explicitly constrained to be a set of unique attributes.

Formally, for each object class O_i with the properties $\{p_{i_1}, p_{i_2}, \ldots\}$, each subset of properties that are constrained to always have a unique value for each object in the class at any given moment is identified.

SIIPS MONITORING DATA MODEL:

Figure 36 shows the identifying properties, and the chosen ruling part for each of the object classes of our model. In some cases, however, it is difficult to decide which set of identifying properties to choose as ruling part. Also, sometimes it is not clear whether a property is "natural" (has a useful correspondence in the real-world situation) or "artificial" (created specifically for use in the ruling part of some object class in the data model). An example is a property VOYAGE-NUMBER of the object class VOYAGES, which has not been included in the properties of VOYAGES discussed so far.

It is clear that voyages of a particular ship are ordered. Hence, it makes sense to talk of the first, second, or last voyage of a ship. However, this order can be easily deduced from the departure date of each voyage. It is unlikely that the user will refer to the n^{th} voyage of a ship, and more likely that the user will refer to the last or next to last voyage, which can be deduced from the voyage number or departure date of the voyage.

object class	identifying properties	ruling part
SHIPS	(1) NAME (2) HULL NUMBER (3) IRCS (4) ID	(4)
SHIP-CLASSES	(1) NAME	(1)
SHIP-OWNERS	(1) NAME	(1)
SHIP-CHARTERERS	(1) NAME	(1)
SHIP-TYPES	(1) NAME	(1)
CARGO-TYPES	(1) NAME	(1)
CARGO-CLASSES	(1) NAME	(1)
PORTS	(1) NAME	(1)
COUNTRIES	(1) NAME	(1)
VOYAGES	(1) SHIP-ID, DEPART-DATE (2) SHIP-ID, VOYAGE-NUMBER	(2)
LEGS	(1) SHIP-ID, VOYAGE-NUMBER, LEG-NUMBER (2) SHIP-ID, VOYAGE-NUMBER, LEG-DEPART-DATE	(1)
STOPS	(1) SHIP-ID, VOYAGE-NUMBER, STOP-NUMBER (2) SHIP-ID, VOYAGE-NUMBER, STOP-ARRIVE-DATE	(1)

Figure 36 Ruling parts for the object classes

This ordering of voyges should be represented in the data model. The user can then use the functions of the SMQL that apply on *ordered domains* (see Section 4.3.6) in a straightforward manner. Since one of the purposes of the model is to allow the user convenient access to the information in the database, we will include a voyage number attribute in our model.

It is also plausible that the user would refer to the first, second, or n^{th} leg of a voyage. Likewise, it is plausible that the user would refer to the first, second, or n^{th} stop of a voyage. However, this order is again easily derivable from the daparture time for each leg. Since the goal of the basic model is minimum redundancy, we must decide which properties to represent in the basic model, and take note of other properties derivable from them to represent those in the augmented model.

5.6.6 Identify the relationships between object classes

The relationships between the object classes are identified, and their structural properties determined by examining the real-world situation. Additional constraints that apply to the object classes or the relationships that do *not* fall into the structural properties are also identified. In addition, each relationship is classified as either static (not expected to change frequently) or dynamic (expected to change frequently). This distinction can be useful for system implementation.

Relationships are identified in most cases by examination of the properties of each object class, and then the determination *for each property* whether it serves to relate the object class to another object class (possibly the same one). Hence, what we do is determine which properties are relating properties of the object class. In some cases, a relationship may exist even though the relating properties were overlooked in the prior design stages, so *it is not sufficient to examine object classes for relating properties*. The ruling part defined in the previous step is used in the structural model representation of the relationship.

Formally, for each object class O_i with the properties $\{p_{i_1}, p_{i_2}, \ldots\}$, each subset of properties is examined to determine whether they relate this object class to another object class *via a relationship*. For each such set of relating properties, the corresponding relationship is identified, and its structural properties—the cardinality and the dependency (see Sections 2.2.2 and 5.3.2.1)—are determined by examination of the real-world situation.

In some cases, new (relationship) object classes may be created whenever a relationship among more than two object classes exist. Note that at this stage, we do not worry about redundancy. Hence, some relationships that we identify may be derivable from others. We create the minimum redundancy basic model in the following steps.

SHIPS MONITORING DATA MODEL:

In our model, the relationships that are identified are shown in Figure 37, along with their cardinality and dependency properties. When we give the dependency property, partial A on B means that the object class given first in the relationship is dependent on the object class given second. Partial B on A means the reverse.

5.6.7 Design of the minimal relations of the basic data model

In this stage, the basic structural model is designed. Properties that are derivable from others are not represented in the basic model, but are noted so they will be represented in the augmented model (see Section 5.6.9).

So far in our design, we did not require that a property be represented in only one place. Now, the data model designer creates the basic data model, a minimum redundancy model, by identifying properties derivable from one another. Derivable properties do not have to be derived from properties of objects in the same object class, but can also be derived *from properties of related object classes*. For each set of properties that are derivable from one another, only one is chosen for inclusion in the basic model, such that the other properties can be derivable from it.

The next step is to define the relations of the basic structural model, based upon the properties chosen to satisfy the minimum redundancy. Following that, the relationships between object classes are represented by connections so that the structural properties of the relationships are correctly represented (see Section 5.6.8). Relationships may also be derivable from other relationships, so not all relationships will be represented by connections in the basic model.

relationship	*name of relationship*	*cardinality*	*dependency*
SHIP:SHIP-CLASS	SHIP CLASS	N:1	total (1)
SHIP-CLASS:SHIP-TYPE	TYPE OF SHIP CLASS	N:1	partial A on B
SHIP:SHIP-TYPE	TYPE OF SHIP	N:1	partial A on B
SHIP:COUNTRY	COUNTRY OF REGISTRY	N:1	partial A on B
SHIP:OWNER	OWNER OF SHIP	N:1	partial A on B
OWNER:COUNTRY	COUNTRY OF OWNER	N:1	partial A on B
SHIP:COUNTRY	OWNER COUNTRY OF SHIP	N:1	partial A on B
SHIP:VOYAGE	PAST SHIP VOYAGES	1:N	partial B on A
SHIP:VOYAGE	CURRENT SHIP VOYAGE	1:1	partial B on A
VOYAGE:LEG	PAST LEGS OF VOYAGE	1:N	partial B on A
VOYAGE:LEG	CURRENT LEG OF VOYAGE	1:1	partial B on A
VOYAGE:CHARTERER	SHIP CHARTERER FOR VOYAGE	N:1	partial A on B
LEG:CARGO-TYPES	CARGOES ON BOARD FOR LEG	M:N	no-dependency
LEG:PORT	SOURCE PORT OF LEG	N:1	partial A on B
LEG:PORT	DESTINATION PORT OF LEG	N:1	partial A on B
VOYAGE:STOP	STOPS OF A VOYAGE	1:N	partial A on B
STOP:CARGO-TYPE	CARGOES UNLOADED AT STOP	M:N	no-dependency
STOP:CARGO-TYPE	CARGOES LOADED AT STOP	M:N	no-dependency
SHIP-CLASS:CARGO-CLASS	CARGOES FOR SHIP CLASSES	M:N	no-dependency
SHIP:CARGO-CLASS	CARGOES SHIPS CAN CARRY	M:N	no-dependency
CARGO-CLASS:CARGO-TYPE	CLASSES OF CARGOES	1:N	total (1)

(a) Static relationships

SHIP:CARGO-TYPE	CURRENT CARGOES ON BOARD	M:N	no-dependency
SHIP:PORT	CURRENT SHIPS AT PORT	N:1	no-dependency
SHIP:CHARTERER	CURRENT CHARTERER OF SHIP	N:1	no-dependency
SHIP:PORT	CURRENT DESTINATION OF SHIP	N:1	no-dependency
SHIP:PORT	CURRENT SOURCE OF SHIP	N:1	no-dependency

(b) Dynamic relationships

Figure 37 Relationships of the ships monitoring data model

In the case of object classes that are derivable from one another, we try to include the fewest number of object classes that cover all the properties of the desired object classes, and represent them in the basic structural model. In most cases, the ones that are considered most "natural" will be best.

Sometimes, as in the case of legs and stops, it is not clear which concept to represent in the basic model, and which is to be derived. We leave this to the discretion of the designer. One possible guideline would be to represent the concept that is likely to be used more often in the basic model. Another possibility is to represent both concepts, and include some basic properties in each. In the latter case, it may be necessary to represent some data redundantly, and include additional constraints to ensure the correspondence of the redundant structures. Note that as far as the user is concerned, basic and derived structures in the data model are equivalent, since they are used in the same way. The only exception to this rule is that some derivable concepts are not updatable.

At this point, we can deal only with relations, connections, attributes, and domains of the structural model that represent the object classes, relationships, and properties of the real-world situation. This is because the real-world information is now being encoded into a formal model that is used as a basis for implementing the database system.

The design goal for the basic data model is to represent each attribute that represents a domain of values in only one relation, except where the attribute is used to define a connection. This leads to minimum redundancy in the sense of minimum repetition. Minimum redundancy reduces the constraints on the basic model considerably, since whenever an attribute is represented redundantly, a constraint is needed to specify that the two attributes are the same, and hence that corresponding values for the attributes should correspond in all database instances that correspond to that model. Once the basic data model is defined, all derivable concepts are added in the augmented model as discussed in Section 5.6.9.

Another way of looking at the minimum redundancy goal of the basic data model is to try and make every data value in a database that is a *literal interpretation* of the data model appear a *minimum number of times*. By literal interpretation, we mean that every relation is implemented as a single file without any access structures. In a minimal literal interpretation, a primary version of each data value appears once, and only as many other repetitions of this data value exist as are required to represent its connections to other files. In actual implementations, this does not hold because of the use of pointers, indexes, ...etc, in the access paths to records of the file, and because of possible redundancies in the files.

If the number of basic attributes in the data model is m, and the number of connections in the model (that are needed to represent the structural properties of relationships) is n, then the minimum total number of attributes in all relations of the basic model is $m + n$. This is because each connection increases the number of attributes by repeating an attribute in the relation at the other end of the connection. Since some connections are defined by more than one connecting attribute, the actual minimum redundancy number of attributes n_A is

$$n_A = m + \sum_i i \times n_i$$

where n_i is the number of connections in the basic model that are defined by i connecting attributes, and

$$\sum_i n_i = n.$$

We can use the number n_A to check that our basic structural model is a minimum redundancy model, or close to a minimum redundancy model.

Formally, for each object class O_i with the properties $\{p_{i_1}, p_{i_2}, \ldots\}$, we do the following:

(a) Create the main relation R_i for the object class O_i that will be part of the basic data model. We give R_i a name close to the the name of O_i, as discussed in Section 5.5. The attributes of the main relation R_i are the non-repeating properties of the object class O_i that are not derivable from the other properties of O_i, or other properties of object classes related to O_i.

(b) For each repeating property, or set of repeating properties, of the object class O_i that is not derivable from the other properties of O_i, or other properties of object classes related to O_i, create a *nest relation* R_{i_j} *that is owned by* R_i *via an ownership* connection.

(c) For each relationship between O_i and another object class, determine the relating attributes, and design the connections that represent the structural properties—the cardinality and the dependency (see Sections 2.2.2 and 5.3.2.1)—of the relationship. We expand on this point in the next section.

SHIPS MONITORING DATA MODEL:

We analyze the properties defined thus far in our model for duplicates—properties that are derivable from one another—and identify multiple occurrences of properties in different object classes. For each such property, we must decide in which relation the corresponding attribute will belong.

We decide this based upon which relation new values of the attribute are inserted or updated most "naturally". Unfortunately, this concept of naturalness is not well defined, but in most cases, as we shall see, it is easy to determine.

The following duplicates are discovered.

(1) The properties DEPARTURE-PORT and DESTINATION-PORT of SHIPS are the same as the properties *DEPARTURE-PORT* and *DESTINATION-PORT* for the PRESENT-LEGS for that ship. We place the two properties in PRESENT-LEGS.

(2) The property LAST-REPORTED-POSITION of SHIPS is the same as the most *recent* track position of the present leg for that ship. Hence, we must recognize the order of the compound repeating property TRACK-POSITIONS of PRESENT-LEGS by the simple properties (DATE, TIME) so that we can derive LAST-REPORTED-POSITION of SHIPS from TRACK-POSITIONS of the present leg for the ship.

(3) The property NUMBER-OF-SHIPS (in class) of SHIP-CLASSES is a count of the current number of ship objects of each class. Similarily, the property NUMBER-OF-DOCKS of PORTS is a count of the number of docks in a port, and hence is a derived property. The properties MAXIMUM-DRAFT, MAXIMUM-BEAM, and MAXIMUM-LENGTH of PORTS are derivable from DOCKS by specifying the maximum value for all the docks in each port.

(4) All properties of PAST-VOYAGES and PRESENT-VOYAGES are common. We need to separate the objects into two classes for easy representation of the constraint that each ship has at most one present voyage. This also leads to easy access to current information about a ship.

(5) The properties for PRESENT-LEGS, FUTURE-LEGS, and PAST-LEGS are all common except for DEPARTURE-DATE which applies only to PAST-LEGS and PRESENT-LEGS, and ARRIVAL-DATE which applies only to PAST-LEGS. Hence, we have them as subrelations.

(6) The properties for LEGS and STOPS are closely interrelated. Here, we consider a voyage to be composed of n stops, with stop numbers from 1 to n, and $n - 1$ legs, with leg numbers from 1 to $n - 1$. For a particular voyage, the departure and destination ports for leg i of that voyage are the same as the ports for stops i and $i + 1$, respectively. Hence, we represent ports in stops only, and derive the departure and destination ports of legs. Similarly, we represent arrival and departure times for legs only, and derive those for stops. For stop i of a voyage, the arrival time is the arrival time for leg $i - 1$, and the departure time is the departure time for leg i of that voyage.

We hence create the relations shown in Figure 38 for the basic data model. Relations that represent an object class are grouped together, and the main relation for each object class is marked by an asterisk (*).

5.6.8 Desig**n** of the minimal connections of the basic data model

We now carry out the same process of the preceding section, but with respect to the relationships specified in the data model, rather than the object classes as discussed in Section 5.6.7. We want to impose a *minimum of constraints on the basic data model*, so relationships that are derivable from others are not represented in the basic model. They will be represented by derived connections in the augmented model as discussed in Section 5.6.9.

First, we examine the relationships for those derivable from others, and choose a *minimal set of relationships*. All the relationships must be derivable from the chosen minimal set. This set is chosen by an analysis of the semantics of the real-world relationships to recognise relationships that are compositions of other relationships, and hence derivable from other relationships.

object classes	*relations*	*ruling part attributes*	*dependent part attributes*	*relation type*
SHIPS:	(*) SHIP	ID	SHIP-CLASS COUNTRY-OF-REGISTRY OWNER CAPTAIN	
	NAME	SHIP-ID	NAME	lexicon
	HULL-NUMBER	SHIP-ID	HULL-NUMBER	lexicon
	IRCS	SHIP-ID	IRCS	lexicon
SHIP-CLASSES:	(*) SHIP-CLASS	NAME	DRAFT LENGTH BEAM DEAD-WEIGHT GROSS-WEIGHT FUEL-TYPE FUEL-CAPACITY CRUISING-SPEED MAXIMUM-SPEED FUEL-CONSUMPTION CREW-SIZE *SHIP-TYPE*	
	POSSIBLE-CARGO-CLASSES	SHIP-CLASS CARGO-CLASS	WEIGHT-CAPACITY VOLUME-CAPACITY	nest
SHIP-TYPES:	(*) SHIP-TYPE	NAME		
SHIP-OWNERS:	(*) SHIP-OWNER	NAME	NATIONALITY ADDRESS	
SHIP-CHARTERERS:	(*) SHIP-CHARTERER	NAME	NATIONALITY ADDRESS	
CARGO-TYPES:	(*) CARGO-TYPE	NAME	CARGO-CLASS UNIT-OF-MEASUREMENT	
CARGO-CLASSES:	(*) CARGO-CLASS	NAME		
PORTS:	(*) PORT	NAME	COUNTRY LONGITUDE LATITUDE	
	DOCK	PORT-NAME DOCK-NUMBER	MAXIMUM-DRAFT MAXIMUM-BEAM MAXIMUM-LENGTH	
	WAREHOUSE	PORT-NAME WAREHOUSE-NUMBER	CARGO-CLASS CAPACITY	
	CARGOES-IN-WAREHOUSE	PORT-NAME WHAREHOUSE-NUMBER CARGO-TYPE	QUANTITY	
COUNTRIES:	(*) COUNTRY	NAME		
VOYAGES:	(*) VOYAGE	SHIP-ID DEPARTURE-DATE	PAST-OR-PRESENT	
	PAST-VOYAGE	SHIP-ID DEPARTURE-DATE		
	PRESENT-VOYAGE	SHIP-ID DEPARTURE-DATE		
	LEG	SHIP-ID VOYAGE-DEPARTURE LEG-NUMBER	PAST-OR-PRESENT DEPARTURE-DATE	
	TRACK	SHIP-ID VOYAGE-DEPARTURE LEG-NUMBER TIME-REPORTED	LONGITUDE LATITUDE DIRECTION REPORTER	
	PRESENT-LEG	SHIP-ID VOYAGE-DEPARTURE LEG-NUMBER		
	PAST-LEG	SHIP-ID VOYAGE-DEPARTURE LEG-NUMBER	ARRIVAL-DATE	
	CARGOES-ON BOARD	SHIP-ID VOYAGE-DEPARTURE LEG-NUMBER CARGO	QUANTITY	
	STOP	SHIP-ID VOYAGE-DEPARTURE STOP-NUMBER	PORT-NAME DOCK-NUMBER PAST-OR-PRESENT-OR-FUTURE	
	PRESENT-STOP	SHIP-ID VOYAGE-DEPARTURE STOP-NUMBER		
	PAST-STOP	SHIP-ID VOYAGE-DEPARTURE STOP-NUMBER		
	FUTURE-STOP	SHIP-ID VOYAGE-DEPARTURE STOP-NUMBER	EXPECTED-ARRIVAL-TIME EXPECTED-DEPARTURE-TIME	
	CARGOES-UNLOADED	SHIP-ID VOYAGE-DEPARTURE STOP-NUMBER CARGO	QUANTITY	
	CARGOES-LOADED	SHIP-ID VOYAGE-DEPARTURE STOP-NUMBER CARGO	QUANTITY	

Figure 38 **Relations and attributes of the basic data model**

For each relationship in the minimal set, we examine the real-world situation for the structural properties of the relationship, and we represent the relationship it by the appropriate connections as discussed in Sections 5.3.

Formally, for relationship R between two object classes O_i and O_j, we obtain its cardinality and dependency by examining the real-world properties of the relationship. We then choose the appropriate representation in the structural model as discussed in Section 5.3.

SHIPS MONITORING DATA MODEL:

By an examination of the relationships of Figure 37, we find that the following relationships are derivable:

(1) TYPE OF SHIP: derivable from SHIP CLASS and TYPE OF SHIP CLASS.

(2) OWNER COUNTRY OF SHIP: derivable from OWNER OF SHIP and COUNTRY OF OWNER.

(3) CARGOES SHIPS CAN CARRY: derivable from CARGOES FOR SHIP CLASSES and SHIP CLASS.

(4) CURRENT CARGOES ON BOARD: derivable from CURRENT SHIP VOYAGE, CURRENT LEG OF VOYAGE, and CARGOES ON BOARD FOR LEG.

(5) CURRENT CHARTERER OF SHIP: derivable from CURRENT SHIP VOYAGE and SHIP CHARTERER FOR VOYAGE.

(6) CURRENT SOURCE OF SHIP: derivable from the CURRENT SHIP VOYAGE, CURRENT LEG OF VOYAGE, and SOURCE PORT OF LEG.

(7) CURRENT DESTINATION OF SHIP: derivable from CURRENT SHIP VOYAGE, CURRENT LEG OF VOYAGE, and DESTINATION PORT OF LEG.

We can represent all the other relationships in the basic data model. The names of the connections are chosen for easy reference to the data model, as we discussed in Section 5.5. Hence, for each connection we have two names, one in each direction. Figure 40 shows the connections between the relations in graphical form, and the names of the connections are given in Figure 39. The choice of representation for each relationship is based upon the structural properties of each as given in Figure 35.

5.6.9 Augment the basic data model by defining all derived attributes, relations, and connections

All the properties and relationships that were left out of the minimal basic data model in the previous two step are now defined as derived attributes, relations, and connections to form the complete augmented model. The augmented model is then examined, and any additional frequently occurring inter-relation references can be predefined as derived connections, to be easily referred to by the users.

1 (type, classes)
2 (class, ships)
3 (owner, ships)
4 (nationality, ship-owners)
5 (country-of-registry, registered-ships)
6 (charterer, voyages)
7 (nationality, ship-charterers)
8 (voyages, ship)
9 (legs, voyage)
10 (reported-positions, leg)
11 (stops, voyage)
12 (cargoes, legs)
13 (cargoes, legs)
14 (cargoes-unloaded, stops)
15 (cargoes-unloaded, stops)
16 (cargoes-loaded, stops)
17 (cargoes-loaded, stops)
18 (docks, port)
19 (warehouses, port)
20 (country, ports)
21 (cargo-class, warehouses)
22 (current-cargoes, warehouses)
23 (current-cargoes, warehouses)
24 (cargo-class, cargoes)
25 (cargo-classes, ship-classes)
26 (cargo-classes, ship-classes)
27 (dock, stops)

Figure 39 Names of the connections of Figure 40

If for every relation that is the main relation representing an object class we define all the attributes logically connected to it as derived attributes, this corresponds to a universal relation of all attributes that we can reference from this relation. Recall that the model forces all attributes of a relation, whether basic or derived attributes, to have unique names. Hence, no ambiquity ever arises. Also, the same attribute may have a different name in each of these universal relations to reflect its meaning when referenced as an attribute of that particular relation.

These relations are similar to relational "views" as described in the literature [Chea76]. They are a means of natural and easy reference to all attributes logically connected to a relation that represents an object class for a single user. A single data model, such as the one designed in this section, corresponds to an expanded user view, since it is the same user who sees the augmented data model. The structural constraints are represented in the underlying basic model, and are automatically enforced on update. For retrieval reference, this universal relation is more convenient to use. For update, which is generally more predictable and is carried out mainly in well-defined transactions, one should refer to the relations of the basic data model.

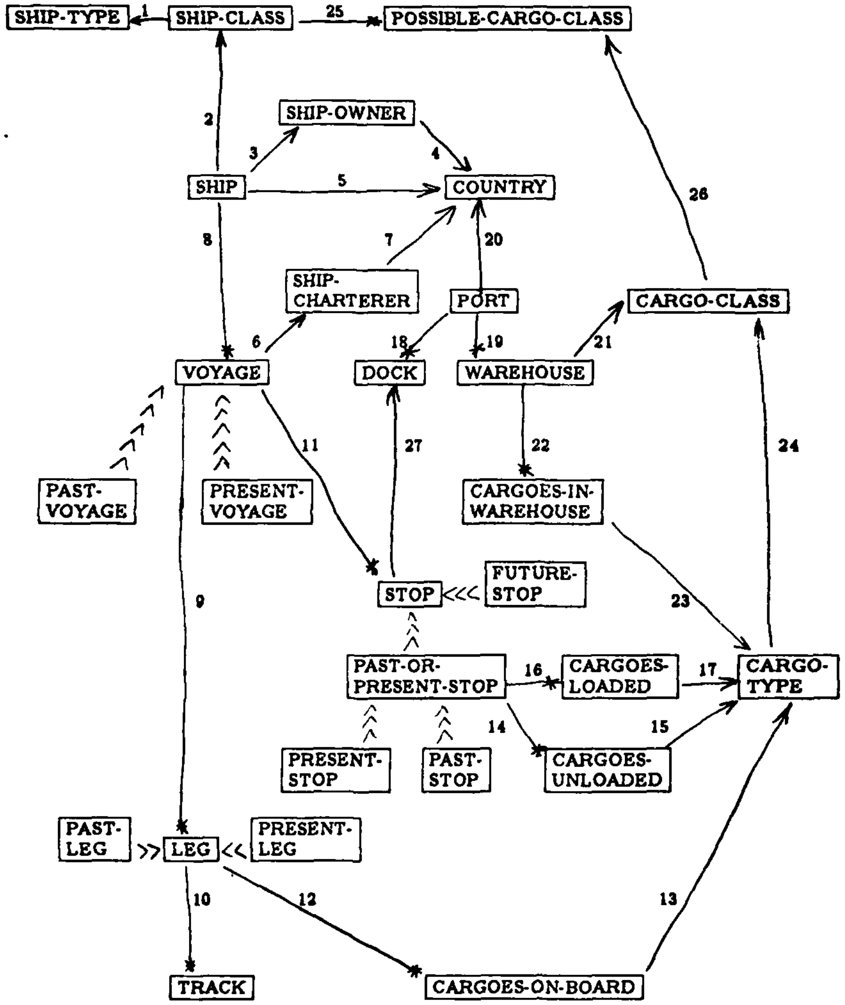


Figure 40 Graphical representation of the SHIPS MONITORING data model

A compromise is to include attributes that are frequently referenced by users from each relation as derived attributes of that relation. Less frequently referenced attributes can then be referenced via the connection names. We will follow this approach in our example.

The statement used to define derived attributes is part of both the Structural Model Definition Language and the Structural Model Query Language, and is presented in Section 3.4.5. A user may define new derived structures using the SMQL as his needs arise. It would be useful in an implementation that the system keep track of all derived definitions, and advise the user of those that are used very infrequently so they may be deleted. This is so that the schema does not become overly clutterred.

SHIPS MONITORING DATA MODEL:

The basic model is now augmented with the derivable attributes, relations, and connections. We give some examples here using the language statement for defining derived structures.

First, we define attributes that are derivable from the connections already specified in the basic model.

(1) The SHIP relation:

(a) All attributes of SHIP-CLASS and SHIP-TYPE are usually referred to from the SHIP relation. Hence, we define them as derived attributes of SHIP.

DEFINE DERIVED ATTRIBUTE DRAFT OF SHIP TO BE
DRAFT OF class OF SHIP

DEFINE DERIVED ATTRIBUTE LENGTH OF SHIP TO BE
LENGTH OF class OF SHIP

...

DEFINE DERIVED ATTRIBUTE TYPE OF SHIP TO BE
NAME OF type OF class OF SHIP

(b) The attributes that describe the current position, cargo, destination, and whether the ship is at some port are also often referred to from SHIP. Hence we define them as derived attributes of SHIP. First, we define the derived connections current-voyage of SHIP, and present-leg of VOYAGE, which connect the SHIP relation to the PRESENT-VOYAGE relation, and the PRESENT-VOYAGE relation to the PRESENT-LEG relation respectively.

DEFINE DERIVED CONNECTION (current-voyage) TYPE OWNERSHIP
FROM SHIP (ID) TO PRESENT-VOYAGE (SHIP-ID)

```
DEFINE DERIVED CONNECTION (current-leg) TYPE OWNERSHIP
FROM PRESENT-VOYAGE (SHIP-ID, DEPARTURE-DATE)
TO PRESENT-LEG (SHIP-ID, VOYAGE-DEPARTURE)
```

```
DEFINE DERIVED CONNECTION (current-stop) TYPE OWNERSHIP
FROM PRESENT-VOYAGE (SHIP-ID, DEPARTURE-DATE)
TO PRESENT-STOP (SHIP-ID, VOYAGE-DEPARTURE)
```

Now, we define the derived attribute POSITION of SHIP, which is a compound attribute composed of the two simple attributes LONGITUDE and LATITUDE. Here, we see an example of the use of ordering defined on the attribute TIME-REPORTED of the TRACK relation. Note that this is in fact the last reported position of the ship rather than its current position. We assume the database is regularly updated, and that ship positions are reported within short intervals to make this difference unimportant.

```
DEFINE DERIVED ATTRIBUTE CURRENT-POSITION OF SHIP TO BE
((LONGITUDE, LATITUDE) OF reported-positions: TIME-REPORTED =
MAXIMUM TIME-REPORTED OF reported-positions OF current-leg
OF current-voyage OF SHIP) OF current-leg OF current-voyage OF SHIP
```

For the cargoes on board the ship, we define a derived attribute CARGO of SHIP. Again, this is a compound attribute, and in addition it is a repeating attribute, since it returns the set of (cargo, quantity) pairs currently on board a ship.

```
DEFINE DERIVED ATTRIBUTE CARGO OF SHIP TO BE
(CARGO, QUANTITY) OF cargoes OF current-leg
OF current-voyage OF SHIP
```

Finally, to specify the subset of ships that are currently at a port, we have to specify all ships that are currently connected to a present stop.

```
DEFINE DERIVED RELATION SHIP-AT-PORT TO BE
GET ship OF voyage OF PRESENT-STOP
```

Note here that the connection (stop, voyage) (connection number 11 in *Figure 40*) is referred to from the subrelation PRESENT-STOP of STOP. This is allowed by the inheritence rule for subrelations. Also note here that the subrelation SHIP-AT-PORT of SHIP only defines a subset of the tuples of SHIP and can refer to all of the attributes of a ship. No attributes are specified in the definition of the derived relation, but implicitly all attributes of SHIP apply to tuples in SHIP-AT-PORT. Hence, SHIP-AT-PORT is an example of a subrelation of the SHIP relation that is not represented in the basic data model, but is defined in the augmented model for easy reference. We can now define additional derived attributes which apply to the subrelation SHIP-AT-PORT of SHIP.

For example, the attributes PORT and DOCK are defined by:

```
DEFINE DERIVED ATTRIBUTE PORT OF SHIP-AT-PORT TO BE
NAME OF port OF dock OF current-stop OF voyage OF SHIP-AT-PORT
```

```
DEFINE DERIVED ATTRIBUTE DOCK OF SHIP-AT-PORT TO BE
NUMBER OF dock OF current-stop OF voyage OF SHIP-AT-PORT
```

Here, we used the derived connection current-stop to locate the unique current stop of a ship.

(c) We can similarly define the connections possible-cargoes, and cargo-classes of SHIP, which give the possible types and classes of cargoes a ship can carry. However, we will assume that this is not a frequently occurring request, and leave it to be specified explicitly via the connection names if desired.

(2) The SHIP-CLASS relation:

(a) The NUMBER-OF-SHIPS attribute of SHIP-CLASS, which gives the number of ships *in a given ship class is derivable by counting the number of ships of that class* that exist in the database.

```
DEFINE DERIVED ATTRIBUTE NUMBER-OF-SHIPS OF SHIP-CLASS
TO BE COUNT ships OF SHIP-CLASS
```

Few references are expected to SHIP-CLASS by the users, and hence this is the only derived attribute we define. Most user requests will refer to SHIP, but SHIP-CLASS was necessary in the basic data model to represent the constraint that ships of the same class share the same values for a number of properties.

(b) The attribute NO-OF-CARGOES of CARGO-CLASS can similarily be defined as the number of cargo types of each cargo class. This derived structure that counts the number of objects of one class connected to an object of another class is quite common.

(3) The PORT relation:

Several attributes are derivable from the DOCK and WAREHOUSE relations connected to the PORT relation. These include the attributes NUMBER-OF-DOCKS, MAXIMUM-DRAFT, MAXIMUM-LENGTH and MAXIMUM-BEAM (allowable at the port), and NUMBER-OF-SHIPS (currently at the port) from the DOCK relation. From the CARGOES-IN-WAREHOUSE relation, we can get the cargoes in a port.

For example, the attribute MAX-DOCK-LENGTH of PORT is defined as:

```
DEFINE DERIVED ATTRIBUTE MAX-DOCK-LENGTH OF PORT TO BE
MAXIMUM LENGTH OF dock of PORT
```

Note that derived attributes can be defined at any time during the operation of the system, and not necessarily during the original design phase of the data model. As the need arises, new derived classes are created and old ones not being used are destroyed. This *corresponds to change in the usage* of the underlying database, and should be noted. If significant changes occur, this may signal the need for restructuring the physical database to provide a better performance for the changing user requirements.

Hence, it is useful to collect information about the usage of each of the derived structures of the model. Such monitoring is *less expensive at the schema level*, and we believe can be effectively used to recognise changes in database usage.

5.7 CONCLUSIONS

In this chapter, we presented the steps taken to design a data model *for some* real-world application using the structural model. The emphasis is on a thorough analysis of the real-world application to extract the relevant rules that apply to object classes and relationships, and represent these rules in the data model for the application.

In Sections 5.2 to 5.4, we showed in general how each of the real-world concepts presented in Chapter 2 can be represented using the structural model.

In Section 5.2, we showed how object classes are represented. We discussed how properties of an object classes, whether simple or compound, repeating or non-repeating, are represented to reflect the fact that the properties are used mainly to describe an object, and hence are owned by the object. We presented the concept of a *main* relation that represents the object class. We then showed how repeating properties are represented by nest relations owned by the main relation, and how duplicate identifying properties are represented by lexicon relations connected to the main relation. Although these properties are represented by attributes that are not in the main relation which represents the object class, we showed how reference to these properties from the main relation is straightforward in the SMQL, so that it is quite similar to the reference to properties that are represented by attributes in the main relation itself.

We then discussed the representation of relationships in Section 5.3. The emphasis was on a correct representation of the structural properties of the relationship, and on specifying the rules that maintain the structural properties in the model itself. We showed in Section 5.3.2 that there are a limited number of ways to maintain the structural properties of a relationship, and showed how the structural model specifies the maintenance rules. An important concept is that whether the relationship was represented by two or three relations, the reference to the relationship in the SMQL is the same. Hence, the choice of representation, based on the known structural properties of the relationship, is independent of the way the user refers to the relationship in the query language.

In Section 5.4, we showed how the structural model represents subclasses, and some special types of relationship, and how the query language refers to these representations. For subclasses, the inheritance rule of attributes allows the user to access all attributes and connections of the base relation directly from the subrelation.

We then discussed the guidelines to follow for the choice of attribute, relation, and connection names so that the user interface will appear quite close to the real-world concepts.

Finally, in Section 5.6, we presented the steps to be followed in the design of a data model for a particular application of a group of users. We illustrated the design steps by a fairly complex data model to demonstrate all the concepts involved. We showed how to separate the minimal *basic* data model, that represents the constraints from the real-world, from the *augmented* data model, which includes many derived structures for easy access by the user.

Such a data model is designed for each user group that has some application. In general, many such user groups, with separate real-world applications, will want to use the database system. Some of their data classes will be common. In the next section, we discuss the process of *integrating* the many user data models that represent numerous applications to create an *integrated database model*, which will support all the users of the database.

For the integration process, we consider only the basic data models of the different users, since these basic data models contain the complete data content, and the constraints on the data for each user group. Each user group will see only its own model, and the mappings to the integrated database model are handled exclusively by the database system. The database system also handles all the mappings from the integrated database model to the actual data as it is stored on computer files. The integrated database model is used as a basis for specifying the global data contents for the entire database system.

6 DATA MODEL INTEGRATION

6.1 INTRODUCTION

The design of a data model, discussed in Chapter 5, is a process that applies to each potential user, application, or group of users, of the database system. We call each of these a *database user*. Each database user, aided by some database design expert, will define data and processing requirements. From these requirements, a data model is designed. At the same time, other quantitative expected usage information and performance requirements are collected; we sketched how these mey be used in [WiEl79b], but we do not address this issue here.

Next, the database design experts will integrate these data models to produce an integrated database model. If all data models are identical, an unlikely situation, this integration process is trivial. A more likely situation is the data models will be in agreement on common parts of the situation they represent, but will have different scopes. For example, the data models may differ in the properties they use to represent an object class, or they may represent different subsets of objects of the same object class. In such cases, the data models need not be changed, but the integrated database model should support all the data models correctly. Finally, it is possible that different users may understand a common situation differently, so that the two representations are irreconcilable. In this case, a conflict arises that has to be resolved by a reexamination of the situation by the users to discover which of them is in error. A modification is then made to some or all of the data models. In Section 6.2 we discuss the cases where two data models can differ, and which of these cases can be integrated.

Once the models have been integrated, the global database model will support the database submodels of the different database users. If no change had occurred to a user data model during the integration process, this user's database submodel will be identical to his data model. If some conflict had arisen during integration which resulted in some change to the user data model, the database submodel resulting from the integration process will be different from the original data model to reflect this change. Figure 41 illustrates the integration process.

Another reason for the database submodel differing from the data model, even though the data model does not conflict irreconcilably with the other data models, is to simplify the integrated database model. When different representations of some situation exist, it is worthwhile to reevaluate the data models to see if the users can agree on a common representation, which is then adopted in the integrated database

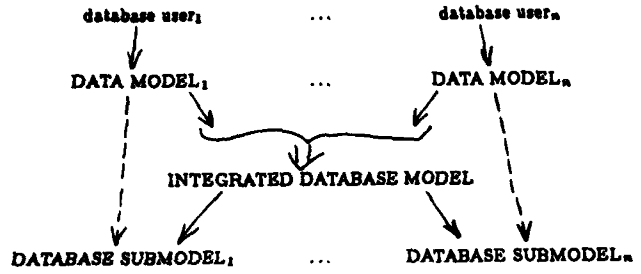


Figure 41 The integration process

model. Hence, the integration process is not carried out idependently, but by consultation and interaction with the users. New insights may arise during the integration process causing a change in the point of view of some users. However, data models do not have to be changed if strong reasons exist for the different representations.

To recap, the terminology we use in our discussion of integration is the following:

(1) A *data model* (or *user model*) is a representation of the requirements of a particular database user or application. The definition of data models for individual users or user groups that expect to use the database is the first step in the design of an integrated database. The design of individual data models was discussed in Chapter 5. In the integration process, we mainly consider the basic data model of each user. Derived representations are only considered if they are part of the basic model of some user, but are derived in data models of other users.

(2) The *database model* is the integrated model created by merging the individual data models. During merging, differences in view of common situations are bound to appear. Some differences can be resolved by creation of an integrated database model that supports all of the views. For unresolvable conflicts, some data models have to be changed, or the integration abandoned with respect to some of the data models. The integrated database model corresponds to the *conceptual schema* of the ANSI/SPARC architecture [Ste75].

(3) A *database submodel* is a user model that is consistent with the integrated database model. Hence, if no conflicts had occurred between a user data model and other data models during integration, the integrated database model will correctly support the user data model, so the database submodel for that user will be the same as the data model. If some conflict had arisen, some differences will exist between the data model and the database submodel, the result of changing the data model in consultation with the user. The database submodels corrrespond to the *external schemas* of the ANSI/SPARC proposal.

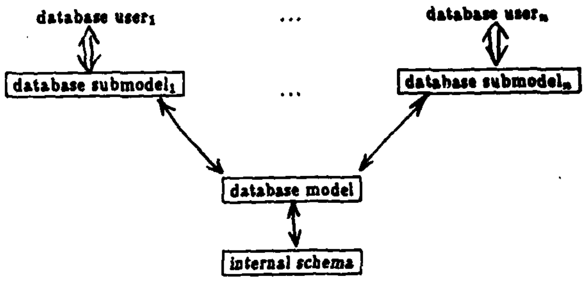


Figure 42 The operational database management system

Once the integration process is completed, the user database submodels and the integrated database model are implemented on the database system. The structure of the operational database management system is shown in Figure 42. Each database user will only see and interact with his or her corresponding database submodel. The user will pose his query and update requests in a query language, such as the SMQL, based on the structures of the database submodel. The system then automatically maps these requests to the structures of the integrated database model, and finally translates the request to the appropriate access operations on the file structures. The file structures and their access paths are described by the internal schema. Any optimisation of the user requests will take place at the internal level.

Note that this architecture supports complete independence of the logical models (the database submodels and the integrated database model) from the physical storage (described by the internal schema). *We can change the file or access structures without* any effect on the logical models. The only changes required are to the mapping from the database model to the internal schema.

New users can be added to the operational system. If the information in their data model is directly derivable *from the current* database model, we only need to define a new submodel, and its mapping to the database model. If the information needed for the new data model is totally contained in the database model, but not directly derivable from it, the database model may have to be slightly changed, and the mapping to the internal schema for the changed structures must be redefined. Finally, if the new user needs to introduce information that is not included in the current database model, we have to change both the database model, and the internal schema—and hence reorganise some of the actual files. This may be an expensive operation and should not be carried out indiscriminately.

In section 6.2, we consider some general concepts of data model integration. In section 6.3 we consider the integration of relations *from* different data models that represent the same real-world object class. In section 6.4, we show how to integrate two *different representations* of a relationship between the same two real-world object classes. Finally, in Section 6.5 we give a short algorithm for integration which handles the basic cases.

6.2 CONCEPTS OF INTEGRATION

The data models we *integrate represent* real-world situations that partially overlap, otherwise there is no need for integration. *Hence we expect* to find relations in the separate data models that are the *main* relations (see Section 5.2) which represent the same object classes. The first phase of integration is to recognise such relations. This is *not* always a simple task, since different data models may use different names for the main relations *which represent* the same object class. If the naming conventions of Section 5.5 are adhered to, the process of *name comparison* to discover relations that represent the same object class can detect most of these cases.

Recognition of *relations that represent* the same object class in different data models is based on matching the ruling parts of the relations, since the ruling parts define the correspondence to real-world object classes, and values of *ruling part* attributes define *correspondence* to real-world objects. The relation names and the ruling part attributes names can *provide an initial* hint to such correspondences, especially if the naming guidelines of Section 4.5 are followed. However, to ensure the correspondence, an examination of the domain definitions *of the ruling part* attributes is required. A partial or total *match* of domains can establish the correspondence. This may require *examination* of existing data for the domains, and referal to users for confirmation.

Since there are often lexicons for *ruling parts*, we consider these lexicons when searching for relations that represent the same object class, since different users can have *different ruling* parts that are matched by a one-to-one correspondence. An example is given in section 6.3.

The second phase of integration, following the recognition of relations that represent *the same object* classes, is the recognition of *differences* in the representation of the object classes in the data models.

These differences are of three types:

(1) Representation of *different properties of the* same object class: this is reflected in different dependent part attributes in, and nest relations *owned* by, the main relations that represent the same object class. Hence, users have different projections of the *attributes* that represent the properties of the object classes.

(2) Representation of different subsets of objects of the same object class: this is reflected in different tuples in the main relations that represent the same object class. Here, users have different restrictions on the tuples that represent the objects *of the* class.

(3) A combination of (1) and (2).

We will cover integration of those cases in Section 6.3.

The final *phase* is to integrate the *representation* of relationships between object classes. As shown in Section 2.2, relationships between more than two object classes are decomposable into several *relationships* between two object classes. Hence, we only consider such binary relationships in our analysis.

There are five ways to represent binary relationships in the structural model (see Section 4.3.1, Figure 17). Data models may choose to represent the same relationship between the same two entity classes differently, according to their view of the situation. Hence, in this phase of the integration, we create an integrated database model which will support different representations of relationships in the data model. We cover this phase in Section 6.4.

Many data models may have to be integrated into a single database model. To avoid excessive complexity we will analyse the integration of only two data models in detail. Successive integration steps can merge another data model with the database model being built, creating a new database model. Since both data models and database models use the same structural model primitives, this should not pose a problem.

We hence have two data models, data model 1 (*dm1*) and data model 2 (*dm2*). Both data models will include relations that represent some common object classes, as well as other classes of data. We will only look at one object class A in Section 6.3, and two object classes A and B with a relationship between them in Section 6.4. R_a and R_b will denote the main relations that represent object classes A and B in *dm1* and *dm2* . If both representations are the same, clearly there will be no need for any transformation, and the integrated database model (*idbm*) will use the same representation. If representations differ, we create an *idbm* to support both models.

The *idbm* then supports database submodel 1 (*dbsm1*) and database submodel 2 (*dbsm2*), that correspond to *dm1* and *dm2* respectively. In some cases, *dm1* and *dm2* are not changed, so *dbsm1* and *dbsm2* will be identical to *dm1* and *dm2* . In other cases, where conflicts appear, one or both of the data models may be changed, and the corresponding database submodels will reflect those changes. Once the database model is established, it may also be desirable for pragmatic reasons to change a database submodel to achieve a better agreement with the database model.

In many cases, only a subset of the tuples in relation R_a (or R_b) of the *idbm* will correspond to the R_a (or R_b) relation included in *dbsm1* or *dbsm2* . We then use a subrelation to represent this subset, and an identity connection will connect it to R_a (or R_b) of the *idbm* . For example, if R_a in *dbsm1* corresponds to a subrelation of R_a in the *idbm* , we denote this subrelation by R_{a1} in the *idbm* , and R_{a1} will have an identity connection to R_a in the *idbm* . The subrelation R_{a1} of R_a will include only the ruling part attributes of R_a, so that no duplication of information occurs in the *idbm* . All other attributes of R_a are accessed through the identity connection to R_a. Hence, R_{a1} represents the restriction of tuples that correspond to the tuples in R_a of *dbsm1* .

We do not address the problem of authorization of users to perform insertion and deletion. We assume that every database submodel has complete insert, delete, and update authorisation over the part of the database model it represents. Hence, if one submodel, *dbsm1* say, inserts a tuple that does not violate the integrity constraints of *dbsm2* , the tuple is inserted in both. If the tuple violates the integrity constraints of *dbsm2* , it is inserted but remains invisible to *dbsm2* . For deletion, if deletion of a tuple is legal in *dbsm1* , say, but the tuple may not be deleted in *dbsm2* because of integrity constraints, the tuple will be kept in the *idbm* and in *dbsm2* , but will become invisible to *dbsm1* . This is the most general case. If some users have retrieval authorisation, but no update authorisation, the integration is simpler since we do not have to consider the action to be taken when that user performs an update.

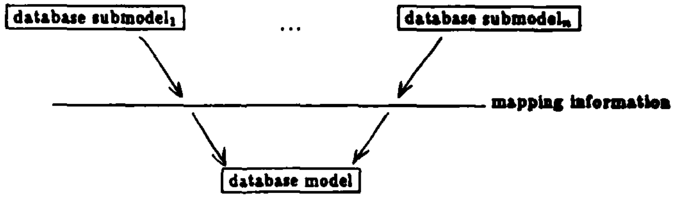


Figure 43 Additional integrity constraints on the operational idbm

After integration, *dbsm1* and *dbsm2* are both supported by the *idbm* . A mapping exists from each submodel to the *idbm* . The mapping (Figure 43) includes additional integrity rules, derived from the integration process, which apply to the *idbm* . These rules are enforced when a database submodel performs an insertion, deletion, or update. We list these additional rules with each case of integration.

The mapping information also includes all transformations from attributes, relations, and connections in each database submodel to the attributes, relations, and connections in the database model. This is a straightforward mapping in most cases. It becomes complex only when a relation in some database submodel includes attributes that are from other relations in the *idbm* , and the submodel requires update authority.

The mapping for a particular submodel is used when that submodel attempts to access the database. The request is expressed in terms of the database submodel, and is automatically translated by the system into the structures of the database model. We will give these mapping rules when we discuss the numerous integration cases. It is very important to note that a user will at no time access the integrated database model. The integrated database model structures may appear unduly complex, but this is so in order to facilitate the statement of the mappings for the user database submodels. Since the users themselves never see the integrated database model, the complexity is hidden from them.

There is an important difference between our integrated database model, and the conceptual schema of the ANSI/SPARC architecture (see Section 1.3, Figure 1). The *idbm* in our proposed system is more than a model of the real-world as in the ANSI/SPARC conceptual schema, since it includes specific structures that model the differences between the database submodels. This is because we view the world as a collection of communities rather than as a massive whole. These structures are used to facilitate the mappings from the *dbsm' s* to the *idbm* .

6.3 INTEGRATION OF DIFFERENT REPRESENTATIONS OF OBJECT CLASSES

In this section, we show how differences in the representation of object classes can be resolved. We differentiate between attribute differences (Section 6.3.2) and tuple differences (Section 6.3.3). The general case can be a combination of both attribute and tuple differences. First, we must recognise the main relations that represent the same object class in *dm1* and *dm2*. We discuss this in the following section.

6.3.1 Recognition of relations that represent the same object class

This phase of integration requires the recognition of relations included in different data models that represent the same object class. Knowledge of the real-world situations being modelled is helpful to match relations that represent the same object class but have different names for relations and ruling part attributes. Adherence to the naming conventions of Section 5.5 will make name comparisons more effective.

If we carry out the data model design carefully, as discussed in Section 5.6, and keep the results of the different design steps, we will have a record of which relations in the data model represent which object classes. We can then compare object class descriptions directly, and know the relations that represent the object class from the data model design documentation.

We must then formally establish that the relations from *dm1* and *dm2* represent the same object class. The domain definitions of the ruling part attributes of the two relations are compared, and a partial overlap or total match of the two ruling part domains will verify the equivalence the relations. In this comparison, we must consider lexicons of ruling parts, as we now discuss.

Some models may include lexicons of ruling parts for some of the relations in the model. Recall that such lexicon relations are used to define a one-to-one correspondence between two attributes, each of which represents an identifying attribute of the object class (see Section 3.2.3). Hence, examination of such lexicons is required when matching ruling parts, since the domain of the ruling part of a relation in one data model may match the domain of a lexicon of a ruling part of a relation in another data model.

For example, *dm1* may include a relation EMPLOYEE that contains the attributes (NAME: ADDRESS, HOME-PHONE, OFFICE, OFFICE-PHONE, DEPT), and represents the directory of employees. Data model 2, which represents job information, includes a relation EMP that contains the attributes (NUMBER: AGE, JOB, SALARY, DEPT>, and a lexicon relation EMP-NAME that includes the attributes (EMP-NUMBER: EMP-NAME) (see Figure 44(a)). To recognize that both relations represent the same object class of EMPLOYEES, the integrators must consider both the NUMBER attribute of EMP and NAME attributes from the lexicon relation in *dm2* when matching the ruling part of the EMP relation of *dm2* to the ruling part of the EMPLOYEE relation of *dm1*.

Formally, two relations schemas R_1 and R_2 represent the same object class if

$$\exists K(R_1)\exists K(R_2): DOM(K(R_1)) \bigcap DOM(K(R_2)) \neq \emptyset,$$

where $K(R_i)$ denotes some set of identifying attributes for the relation R_i (either the ruling part of relation R_i, or a set of attributes that is a lexicon of the ruling part of relation R_i), for $i = 1, 2$.

6.3.2 Integration of relations containing different attributes

Now, we consider two relations R_1 and R_2 that represent the same object class in *dm1* and *dm2* respectively. In this section, we consider the case where the dependent part attributes in the two relations are different. Hence, if G(R) denotes the dependent part attributes of relation R, R_1 and R_2 satisfy

$$G(R_1) - G(R_2) \neq \emptyset, \quad \textit{or}, \quad G(R_2) - G(R_1) \neq \emptyset.$$

We first consider the case where one representation dominates the other. Here, *dm1* includes a relation R_1, and *dm2* includes a relation R_2 that represents the same object class as R_1, and contains all the dependent part attributes represented in R_1, plus some additional dependent part attributes ($G(R_1) \subseteq G(R_2)$). The *idbm* will include a relation R that contains the set of dependent attributes represented in R_1, and a subrelation R' of R that contains the dependent part attributes represented in R_2 but not in R_1. The tuples in R of the *idbm* correspond to the R_1 tuples in *dbsm1*, while the subset of tuples in R' of the *idbm* correspond to the R_2 tuples in *dbsm2*.

When *dbsm1* inserts a tuple in R_1, it is only inserted in R of the *idbm*, since it does not contain the dependent part attributes of R'. The tuple is then only visible to *dbsm1*. When *dbsm2* inserts a tuple, it is inserted in both R and R', since it contains the dependent part attributes of both R and R'. Hence, it will be visible to *dbsm1* also.

The preceding rules hold because we assume that the attributes of R_2 are mandatory, and hence a tuple cannot exist unless it includes valid values for the attributes. If all the attributes that are in R_2 but not in R_1 are optional in R_2, then the preceding rules do not hold. A tuple inserted by *dbsm1* is visible to *dbsm2* also, with unknown values for the additional attributes. Hence, in this special case, we only need a single relation in the *idbm*. We will assume in the following that attributes are mandatory.

The general case is that neither relation R_1 of *dm1* nor R_2 of *dm2* contains the complete set of attributes, but each contains a set of common attributes, and a set of dependent part attributes unique to their model. In this case, a relation R and two subrelations R_1 and R_2 of R are created in the *idbm*. The relation R will contain the common dependent part attributes, while R_1 and R_2 will each contain the set of dependent part attributes only in *dm1* and *dm2*, respectively.

Formally:

$$G(R)_{idbm} = (G(R_1)_{dm1} \bigcap G(R_2)_{dm2}),$$

$$G(R_1)_{idbm} = (G(R_1)_{dm1} - G(R_2)_{dm2}),\ and,\ G(R_2)_{idbm} = (G(R_2)_{dm2} - G(R_1)_{dm1}).$$

Now, when *dbsm1* inserts a tuple, it is inserted in R and R_1 in the *idbm* , but is invisible to *dbsm2* . When *dbsm2* inserts the tuple with the same ruling part value in R_2, the tuple is now inserted in R_2 of the *idbm* , and becomes visible to *dbsm2* . An automatic check is performed to ensure that the common attributes have the same values. Thus, the base relation R in the *idbm* ensures the integrity of common data values. Again, if all the attributes in R_1 only (or R_2 only) are optional, we can do away with R_1 (or R_2) of the *idbm* .

An example is shown in Figure 44. Figure 44(a) shows *dm1* and *dm2* before integration, and Figure 44(b) shows the result of the integration. In Figure 44(b), when *dbsm1* inserts an employee tuple, it is inserted in R and R_1, but is invisible to *dbsm2* , and similarly for R_2. The lexicon relation only references R_2, since it is only represented in *dbsm2* .

It can be worthwhile to consider changing database submodels by making the users aware of a few additional attributes to simplify the integrated database model. In this case, the user is made aware of these additional attributes.

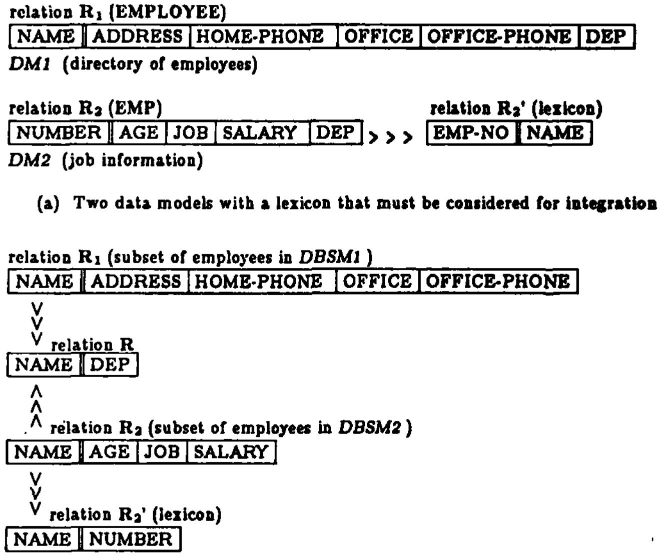
6.3.3 Integration of relations that represent different subsets of the same object class

In the previous section, we saw that when relations have different dependent part attributes that are mandatory, the integrated database model must represent two subsets of tuples, one for each of the data models. It is also possible that each of the two data models includes a relation that represents the same object class, and the same dependent attributes of the object class, but each model represents explicitly a different set of objects from that class. In this case, the two relations will include different sets of tuples, possibly overlapping.

We first consider the case where one data model, *dm1* , includes a relation R_1, and *dm2* includes a relation R_2 that includes a subset of the tuples of R_1. The *idbm* then will include a relation R and a subrelation R' of R to represent the tuples in R_2 of *dbsm2* . This subrelation R' only contains the ruling part attributes of R to define the identity connection from R' to R. The subrelation R' may be a restriction subrelation if the subset of tuples in R' is determined by attribute values in R, or a non-restriction subrelation if the subset of tuples in R' is determined externally by the users, independent of the model.

The general case, is that the relations in *dm1* and *dm2* partially overlap. Then *dm1* with relation R_1 and dm2 with relation R_2 represent the same object class, but the tuples in both relations follow the constraints that

$$R_1 \bigcap R_2 \neq \emptyset,\ R_1 - R_2 \neq \emptyset, and,\ R_2 - R_1 \neq \emptyset,$$


relation R₁ (EMPLOYEE)
NAME | ADDRESS | HOME-PHONE | OFFICE | OFFICE-PHONE | DEP
DM1 (directory of employees)
relation R₂ (EMP)
NUMBER | AGE | JOB | SALARY | DEP
>>>
relation R₂' (lexicon)
EMP-NO | NAME
DM2 (job information)
relation R₁ (subset of employees in DBSM1)
NAME | ADDRESS | HOME-PHONE | OFFICE | OFFICE-PHONE
relation R
NAME | DEP
relation R₂ (subset of employees in DBSM2)
NAME | AGE | JOB | SALARY
relation R₂' (lexicon)
NAME | NUMBER

(a) Two data models with a lexicon that must be considered for integration

(b) Result of the integration

Figure 44 Integration of relations that represent different attributes (with lexicon)

as shown in Figure 45.

The *idbm* then includes a relation $R = R_1 \bigcup R_2$, and two subrelations of R, R_1 and R_2 . Again, R_1 or R_2 could be restriction or non-restriction subrelations.

We now give two examples. In the first example, one relations includes the other. Let *dm1* (for the payroll department of a company) represent all employees of the company in an EMPLOYEE relation, and let *dm2* (for the sales department of the company) include the relation SALES-FORCE, the employees that work for the sales department. The *idbm* then includes an EMPLOYEE relation, with a subrelation SALES-FORCE. If the EMPLOYEE relation contains a DEPARTMENT attribute, the subrelation SALES FORCE would be a restriction subrelation, restricting the DEPARTMENT attribute to the value *sales*. If EMPLOYEE does not contain a DEPARTMENT attribute, SALES-FORCE would be a non-restriction subrelation. In either case, after integration, *dbsm2* is only allowed access to tuples in the subrelation, but could still access all their attribute values from the base relation, while *dbsm1* is allowed access to all employee tuples.

AD-A091 384 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 9/2
ON THE DESIGN, USE, AND INTEGRATION OF DATA MODELS.(U)
MAY 80 R A EL-MASRI MDA903-76-C-0206
UNCLASSIFIED STAN-CS-80-801 NL

2 OF 2
AD A
091384

END
DATE
FILMED
12-80
DTIC

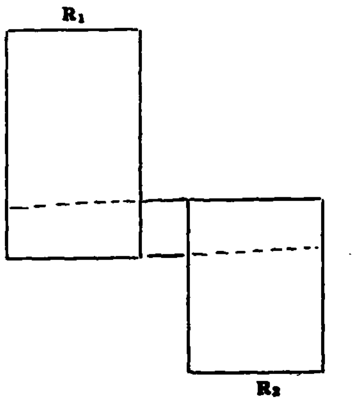


Figure 45 Overlap of tuples in R_1 and R_2

The second example is that of the general case. Referring to a university database, *dm1* (which represents the computer science department of the university) includes a relation PROFESSOR which represents the object class of professors in the computer science department. *dm2* (representing information about permanent faculty) includes a relation PROFESSOR which represents the class of professors who have tenure, while the *idbm* would then include a relation PROFESSOR, and two subrelations of PROFESSOR, CSD-PROFESSOR and TENURED-PROFESSOR. Each database submodel is allowed access to his subset, and the base relation assures integrity of common data in the database.

The most general case is that the relation in each data model contains a set of attributes common to both relations, and a set of attributes that are only applicable to the set of tuples in each relation, as discussed in Section 6.3.2. Additionally, a semantic destinction can be made between the set of tuples in each data model. Then, the integrated database model will include a base relation that contains the common attributes, and the tuples that are common to both data models. Two subrelations, each containing the additional attributes of one of the data models, will represent the tuples in each database submodel. An example is shown in Figure 46.

Hence, we see that integration of two relations that represent the same object class always involves identification of the set of tuples that each data model will see. This is true when the relations differ in their attributes and when they explicitly represent different subsets of tuples. In general, both cases can occur simultaneously.

relation R_1 (PROFESSOR)   (DEPARTMENT = *Computer Science*)

NAME	ADDRESS	HOME-PHONE	OFFICE	OFFICE-PHONE	RANK	TENURE

DBSM1 (computer science department)

relation R_2 (PROFESSOR)   (TENURE = *Yes*)

NAME	BIRTH-DATE	RANK	SALARY	DEPARTMENT

DBSM2 (permanent faculty)

relation R_1 (CSD-PROFESSOR) [restriction: DEPARTMENT = *Computer Science*]

EMP-NAME	ADDRESS	HOME-PHONE	OFFICE	OFFICE-PHONE

∨
∨
∨ relation R (PROFESSOR)

NAME	DEPARTMENT	RANK	TENURE

∧
∧
∧ relation R_2 (TENURED-PROFESSOR) [restriction: TENURE = *Yes*]

EMP-NAME	BIRTH-DATE	SALARY

IDBM

Figure 46 Example of the general case of integrating two relations

6.4 INTEGRATION OF DIFFERENT REPRESENTATIONS OF RELATIONSHIPS

We now consider the integration of two data models, each of which represents a relationship between the same two real-world object classes. Hence, we assume that we have two data models, *dm1* and *dm2*, and that both data models represent two object classes A and B, and a relationship between them. We will denote the relations that represent entity classes A and B by R_a and R_b respectively. If the representation of the relationship between A and B involves a third relation (association, nest of references, or *primary connecting relation*), we will designate it R_{ab}.

There are five ways of representing a relationship between two entity classes in the structural model (see Section 5.3.1, *Figure 18*): two representations involve two relations—the reference and the nest—and three representations involve three relations—the association, the nest of references, and the primary. The number of possible cases for combining different representations pairwise is $2 \times (5+4+3+2+1) = 30$. These 30 cases can be conceptualized as shown in Table 6, where we have two cases to consider *for* each box *in or* above the diagonal. We remove 5 cases where the representation is identical in both data models (1 case from each box along the diagonal), and (5 + 4) cases because the association and *primary representations* are *symmetric* with respect to R_a and R_b (one case from each box along the association and primary columns). Then 16 cases remain to be considered, as shown in Table 7. *If* we consider the cases starting with all cases that involve an association, then all remaining cases that involve a nest of references, and so on with the cases that remain with reference, nest, and primary, we get: 4 cases with the association, 6 with nest of references, 4 with reference connection, and 2 with ownership connection.

dm2 *dm1*	Association relation	Nest of *references*	Reference connection	Ownership connection	Primary
Association relation	2 cases	2 cases	2 cases	2 cases	2 cases
Nest of references		2 cases	2 cases	2 cases	2 cases
Reference connection			2 cases	2 cases	2 cases
Ownership connection				2 cases	2 cases
Primary					2 cases

Table 6 All possible *cases of combining two representations of a relationship*

dm2 *dm1*	Association relation	Nest of references	Reference connection	Ownership connection	Primary
Association relation		1 case	1 case	1 case	1 case
Nest of references		1 case	2 cases	2 cases	1 case
Reference connection			1 case	2 cases	1 case
Ownership connection				1 cases	1 case
Primary					

Table 7 Non-trivial cases of combining two representations of a relationship

In [WiEl79a] and [ElWi79], we considered the integration of all of these cases. Not all of the cases are likely to occur in practice. However, for completeness, we considered all the combinations in Table 7. Here, we will take a slightly different approach, which we believe is more general and more comprehensible. Rather than just consider all possible combinations of different representations, we consider why representations may differ, and what this implies.

We first consider different *representations of a relationship* that has the same cardinality and dependency in both *dm1* and *dm2* in Section 6.4.1. In Section 6.4.2, we consider relationships that have the same cardinality in *dm1* and *dm2*, but have different dependencies, and in Section 6.4.3 we consider relationships that have different cardinality in *dm1* and *dm2*. The first example (Section 6.4.1.1) is used to explain our notation in detail. We illustrate some of the integration cases with examples that can occur in practice.

6.4.1 Integration of relationships of the same cardinality and dependency

Consider a relationship A:B between two real-world object classes A and B that is represented in both *dm1* and *dm2*. In this section, we consider the case where the two data models represent the relationship with the same cardinality and the same dependency *properties*. *For* example, both data models may represent the relationship to be a no-dependency of cardinality *M:N*.

If both data models represent the relationship using the same structure in their respective data models (for example, both data models represent the relationship as an association), and furthermore, if the relations R_a and R_b that represent object classes A and B are identical in *dm1* and *dm2*, then no difference exists, and the *idbm* will use an identical representation. In the following, we consider how a relationship of the same cardinality and dependency can differ in two representations.

6.4.1.1 Integration of association and nest of reference

Now consider the case where one of the data models, *dm1* , represents the relationship as an association, and the other data model, *dm2* , represents the relationship as a nest of references. Assume both representations are of cardinality *M:N* and no-dependency.

This case is shown in Figure 47, where R_a and R_b are the relations that represent object classes A and B respectively, and R_{ab} is the connecting relation as discussed in section 6.1.

In this case, the only difference between *dm1* and *dm2* is that *dm1* can freely delete tuples from R_b, while in *dm2* deletion of tuples from R_b is restricted by referencing tuples from R_{ab}. As we discussed, we assume *dbsm1* and *dbsm2* will both have full insert, delete, and update capability within their database submodel.

The *idbm* is derived from *dm1* and *dm2* so that in the operational database system, it can support the update capabilities for dbsm1 and dbsm2 . Now consider the *idbm* shown in Figure 47. Tuples in R_a and R_b of the underlying database will be visible to both *dbsm1* and *dbsm2* . However, once deletions occur, it is possible that some tuples are deleted from R_b by *dbsm1* (the association representation) which would violate the reference connection from R_{ab} to R_b in *dbsm2* . This is because *dbsm1* allows deletion of R_b tuples that are connected to R_{ab} tuples, and automatically deletes the connected tuples in R_{ab} since the *ownership* connection uses the DLT (delete related tuples) rule to maintain the consistency imposed by the connection. The reference connection uses the PD (prohibit deletion) rule, so the connecting tuples in R_{ab} must be explicitly deleted before the connected tuple in R_b is deleted (see Section 4.3.2).

Hence, we create two subrelations, R_{b1} and R_{ab1}. Those subrelations represent the tuples in R_b and R_{ab} of dbsm1 . Tuples in R_b and R_{ab} of the *idbm* include some tuples deleted from *dbsm1* , but not deleted from R_b and R_{ab} in the *idbm* due to the deletion constraint of the reference connection in *dbsm2* . These tuples are invisible to *dbsm1* .

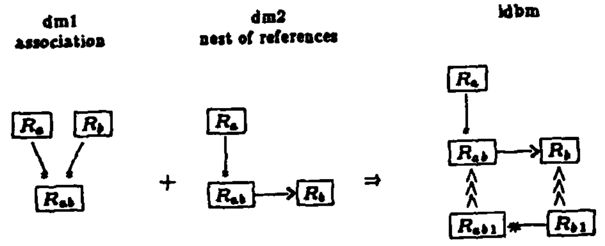


Figure 47 Integration of association and nest of references

The database submodels now obey the following rules. Insertion and deletion in R_a from either *dbsm1* or *dbsm2* is unrestricted, as is deletion of R_{ab} tuples, and unreferenced R_b tuples. If *dbsm1* deletes a referenced R_b tuple (*dbsm2* may not perform such a deletion), it is only deleted from R_{b1}, and the owned tuples are automatically deleted from R_{ab1}. These rules reflect accurately the constraints imposed by the views represented in the original data models.

Because of the additional relations R_{b1} and R_{ab1} in the *idbm* , we need additional rules in the mapping from the submodels to the *idbm* . These rules are needed to specify the cases when an insertion or deletion in one of the submodels requires more than one insertion or deletion in the *idbm* .

For brevity, we will use the following notation for our discussion of integration cases. We first list the differences between the two data models, then list the additional rules that are applied by the *mapping from* the database submodels to the integrated database model. When listing these additional constraints, ("relation name") will mean: do the insertion or deletion specified on "relation" if allowed by the integrity constraints of the idbm .

Hence, in this notation, we would write:

Differences:

dm1 may freely delete tuples from Rb, while in *dm2* , deletion of Rb tuples is restricted by the *reference* connection.

Additional constraints:
dbsm1 :
insert: (1) R_b -: R_b,P_{b1}, (2) R_{ab} -: (R_{ab},R_{ab1}).
delete: (1) R_b -: (R_b),R_{b1}.

dbsm2 :
insert: (1) R_b -: R_b,R_{b1}, (2) R_{ab} -: (R_{ab},(R_{ab1})).

The relation name to the *left* of the "-:" refers to the database submodel, while those to the right refer to the *idbm* . We only list the cases which need additional *control rules from the mapping.* Insert in R_a of *dbsm1* hence means insert in R_a of the *idbm* , since it is not listed. In dbsm1 , insert in R_b requires insertion of the tuple in both R_b and R_{b1} of the *idbm* . Insert in R_{ab} requires insertion in (R_{ab},R_{ab1}) in the *idbm* . The () brackets mean: perform the request if the integrity check of the *idbm* will allow it, here if both owner tuples exist in R_a and R_{b1}.

In *dbsm2* , insert in R_{ab} requires insertion in (R_{ab}, (R_{ab1})), which means: insert the tuple in R_{ab} if the integrity check of the *idbm* holds (here both the owner tuple in R_a and the referenced tuple in R_b exist), then insert the same tuple in R_{ab1} (if the other owner tuple exists in R_{b1}).

In general, whenever one data model uses an ownership connection, and the other data model a reference connection, the model using the ownership connection represents

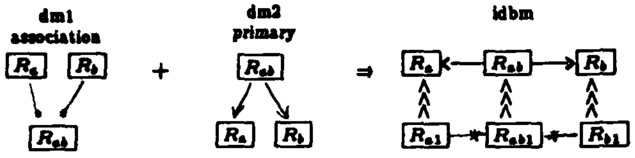


Figure 48 Integration of association and primary

only a subset of the tuples represented in the other model. This is because deletion of tuples in the referenced relation is restricted by references. In most cases, one can trace the difference in representation to a semantic difference between the two relationships.

We now continue our discussion of integration of relationships of the same cardinality and dependency. An *M:N* cardinality can be represented by an association, a nest of references or a primary connecting relation. We discussed above the integration of association and nest of ;eferences, and now briefly cover the other cases.

6.4.1.2 Association and primary

Differences:

1 In *dm2* , deletion of R_a and R_b is restricted by references

2 In *dm1* , deletion of an R_a or R_b causes automatic deletion of the tuples connected to it in R_{ab}.

Additional constraints:
dbsm1 :
insert: (1) R_a -: R_a, R_{a1}, (2} R_b -: R_b, R_{b1}, (3) R_{ab} -: R_{ab}, R_{ab1}.
delete: (1) R_a -: (R_a), R_{a1}, (2) R_b -: (R_b), R_{b1}.
dbsm2 :
insert: (1) R_a -: R_a, R_{a1}, (2) R_b -: R_b, R_{b1}, (3) R_{ab} -: R_{ab}, R_{ab1}.

This case is a good demonstration of the integration where one data model uses an ownership connections, and the other data model uses reference connections. Here, *dm1* , which uses the ownership connections, has unrestricted deletion of tuples from R_a and R_b. Hence, tuples deleted by *dbsm1* which could not be deleted from *dbsm2* because of the reference connection constraint are only deleted from R_{a1} and R_{b1}, but remain in the database, visible only to *dbsm2* .

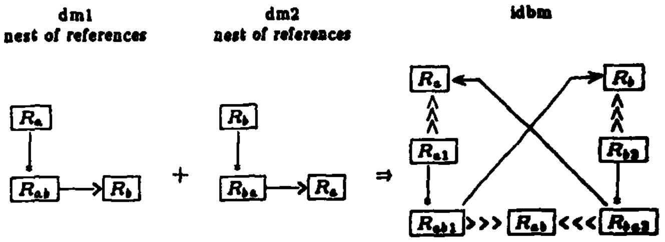


Figure 49 Integration of nest of references and nest of references

6.4.1.3 Nest of references and nest of references

Because the nest of references representation is not symmetric with respect to R_a and R_b, we must consider this case. Associations and primary representations are symmetric since the association uses two ownership connections, and the nest of references uses two reference connections. The nest of references uses one ownership, and one reference connection.

Differences:

(1) Deletion of R_b (R_a) is restricted in dm1 (dm2).

(2) Deletion of R_a (R_b) in dm1 (dm2) requires deletion of owned tuples in R_{ab} (R_{ba}).

Additional constraints:
dbsm1 :
insert: (1) R_a -: R_a,R_{a1}, (2) R_b -: R_b, R_{b2}, (3) R_{ab} -: (R_{ab},R_{ab1},(R_{ba2})).
delete: (1) R_a -: (R_a),R_{a1},(R_{ab}), (2) R_b -: (R_b,R_{ab}).

dbsm2 :
insert: (1) R_a -: R_a,R_{a1}, (2) R_b -: R_b, R_{b2}, (3) R_{ba} -: (R_{ab},R_{ba2},(R_{ab1})).
delete: (1) R_a -: (R_a,R_{ab}), (2) R_b -: (R_b), R_{b2},(R_{ab}).

Here, when *dbsm1* deletes an R_a tuple that is referenced from R_{ba2} in the *idbm* , it is only deleted from R_{a1}, and remains visible to *dbsm2* . If the tuple is not referenced from R_{ba2}, the tuples in R_{ab} that correspond to those deleted from R_{ab1} (due to the deletion of R_a) are also deleted, since they no longer exist in either R_{ab1} or R_{ab2}. The relation R_{ab} exists to ensure that the tuples associating R_a with R_b are consistent. The reference connections from R_{ab1} and R_{ba2} to R_{ab} are restricted to 1:1 connections.

Similarly, when *dbsm2* deletes an R_b tuple, a symmetrical rule is enforced.

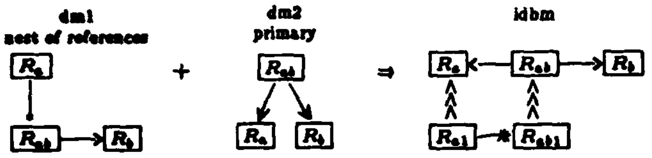


Figure 50 Integration of nest of references and primary

6.4.1.4 Nest of references and primary

Differences:

1 In *dm2* , deletion of R_a tuples is restricted by references

2 In *dm1* , deletion of an R_a tuple causes automatic deletion of tuples connected to it from R_{ab}.

Additional constraints:
dbsm1 :
insert: (1) R_a –: R_a, R_{a1}, (2) R_{ab} –: R_{ab}, R_{ab1}.
delete: (1) R_a –: (R_a), R_{a1}.
dbsm2 :
insert: (1) R_a –: R_a, R_{a1}, (2) R_{ab} –: R_{ab}, R_{ab1}.

We have now considered all possible cases of integrating relationships represented by three relations. We assumed that both relationships are of *M*:*N* cardinality and no-dependency. If the cardinalities or dependencies are further restricted, but are both the same in the two data models, the above integration rules are still valid for the three-relation representations.

Now we consider two-relation representations having the same cardinality and dependency. There are two possible two-relation representations in the structural model: ownership and reference connection. Both these representations represent at most a 1:*N* cardinality and a partial dependency. An ownership connection *from* R_a to R_b represents a relationship A:B of cardinality 1:*N* and partial dependency of B on A. A reference connection *from* R_b to R_a represents a relationship of these same structural properties.

6.4.1.5 Reference and nest

In this case, we assume the relationships are both 1:*N* and partial dependency as discussed above. The integration is also applicable if both relationships are further restricted (for example cardinality to 1:1 or dependency to total) as long as the structural properties of both representations are the same.

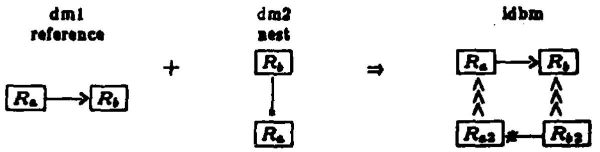


Figure 51 Integration of reference and nest (same cardinality and dependency)

Differences:

(1) In dbsm1, deletion of R_b tuples is restricted by referencing.

(2) In dbsm2, deletion of an R_b tuple requires deletion of related tuples in R_a.

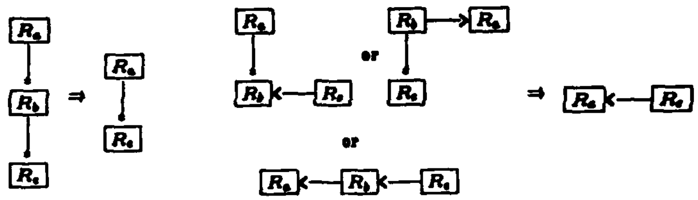
Additional constraints:
dbsm1 :
insert: (1) R_a –: (R_a,(R_{a2})), (2) R_b –: R_b, R_{b2}.

dbsm2 :
insert: (1) R_a –: (R_a,R_{a2}), (2) R_b –: R_b, R_{b2}.
delete: (1) R_b –: (R_b),R_{b2}.

We have now considered the principles of integrating relationships of the same cardinality and dependency. The mapping rules given are derived mechanically, and are to be used as a basis for performing the integration. When integrating specific data models, some semantic analysis of these data models may result in slight modifications to these rules. However, the principles involved are the same, and in most cases the rules will hold. Hence, these rules can always be used as a starting point for the integration, but they may be modified according to additional semantics of a particular situation.

These additional semantics may be due to differences between the object classes represented in *dm1* and *dm2*, as discussed in Section 6.3 (different properties or different subclasses of the object classes). A combination of the techniques discussed in Section 6.3 and above is used to derive the integrated database model.

In Section 6.4.1.7, we give two detailed examples that may occur in practice. First, in the following section, we show how a path of 1:*N* connections can be compacted. This is important because when integrating two data models, one data model may represent a connection between two relations, and the corresponding connection may be a path of more than one connection in another, more detailed, data model.



(a) Combined ownership connection (b) Combined reference connection

Figure 52 Compaction of two 1:*N* connections

6.4.1.6 Combination of connections

The data models we integrate will vary in the level of detail of the information represented in each data model. One aspect of this variation is that one data model will represent numerous properties of an object class, while a second data model may represent only a few properties. This type of variation can be handled by the techniques of Section 6.3. Another type of variation is that some object classes may be completely missing from one of the data models, which may lead to a relationship between two object classes in one data model being represented in more detail as a combination of two or more relationships in another data models.

A more detailed data model, *dm1* (say), may include two relationships A:B and B:C. The other data model, *dm2* , may not include the object class B in its representation, but represent a relationship A:C that is the combination of the two relationships A:B and B:C of *dm1* . An example is given in Section 6.4.1.7.

This type of difference in detail will translate to a relation that is represented in one data model, but not in the other. The less detailed data model will represent a direct connection between two relations that may be represented by two or more connections in the more detailed data model. In this case, the integration is performed based upon the combination of the connections in the more detailed data model. In this section, we show how a path of 1:*N* connections are combined so we can perform the above type of integration.

Figure 52 shows the different combinations of 1:*N* connections. It is straightforward to show from the properties of the connections that whenever a reference connection appears on the path, the compressed connection is a reference connection.

*Definition 23:*A *combination* of two 1:*N* connections, the first from relation R_a to relation R_b, and the second *from relation* R_b *to relation* R_c, *is a combined* 1:*N* connection from R_a to R_c, such that the combined connection exhibits the same

constraints and behaviour on the tuples in R_a and R_c as the two connections exhibit jointly.

Theorem:

The combination of two ownership connections is an ownership connection. The combination of an ownership and a reference connection, or two reference connection is a direct reference connection.

Proof:

The cardinality and dependency properties of an ownership connection from R_a to R_b, or a reference connection from R_b to R_a are the same by definition (cardinality of R_a to R_b is 1:*N*, dependency is partial R_b on R_a). Hence, both of these are maintained by the combined connection. We only need to consider the rules that maintain the dependency property to find the type of the combined connection.

Consider two ownership connections, from R_a to R_b and from R_b *to* R_c. To show that their combination ir an ownership connection, it suffices to show that the behaviour of an ownership connections from R_a to R_b is the same as the behaviour of the two connections.

First consider insertion. Insertion of a tuple in R_a is unrestricted in both cases. Insertion of a tuple in R_c depends on the existence of the owner tuple in R_b, which depends upon the existence of the owner tuple in R_a, so insertion is the same in both cases. Deletion of a tuple from R_c is unrestricted in both cases. Deletion of a tuple *from* R_a causes deletion of the connected tuples in R_b which causes the deletion of the connected tuples *in* R_c. Hence, the behaviour is identical for the combined connection as for the two connections on the path.

The proof is similar for the three other cases. Q.E.D.

Paths of more than two 1:*N* connections (*where the* 1:*N* cardinality is in the same direction) are combined pairwise. When the sequence is made up entirely of ownership connections, the combination will be an ownership connection. If any connection in the sequence is a reference connection, the combination will be a reference connection.

6.4.1.7 Two examples

We now give to examples to illustrate the integration process. The first example integrates two 1:N relationships, one represented by an ownership connection, and the other by a *reference* connection. The second example integrates two M:N connections, one represented by an association, and the other by a nest of references.

Example 1:

Consider a company, or some other large institution, database. We will examine the relationship between employees and their offices in two data models.

Data model 1 is an "employee information" data model. Hence, the EMPLOYEES object class is its most important. A reference connection to the OFFICES object class represents the relationship EMPLOYEES:OFFICES, which is a 1:*N* relationship that relates each employee to his office. We assume each employee is assigned one office. This model is shown in Figure 53(a). The relation BUILDING represents the object class of BUILDINGS, which owns the relation OFFICE.

Data model 2 is a "plant operations and planning" data model. In this model, information about physical buildings and plants is the primary information. Hence, information about employees exists only because employees work in plants to ensure safety regulations, and to assist in planning development. Hence, employees are owned by buildings and offices in this model as shown in Figure 53(b).

The integrated database model is shown in Figure 53(c). In *dbsm2* , it is possible that old buildings scheduled for destruction are deleted from building maintenance before they are actually abandoned. These buildings will not be deleted from *dbsm1* until all employees that were assigned to this building are reassigned to another building.

The additional constraints for this example are given below. They are identical for those given in the integration case of Section 6.4.1.5, except for the additional rules pertaining to relation R_c, since we have this additional relation.

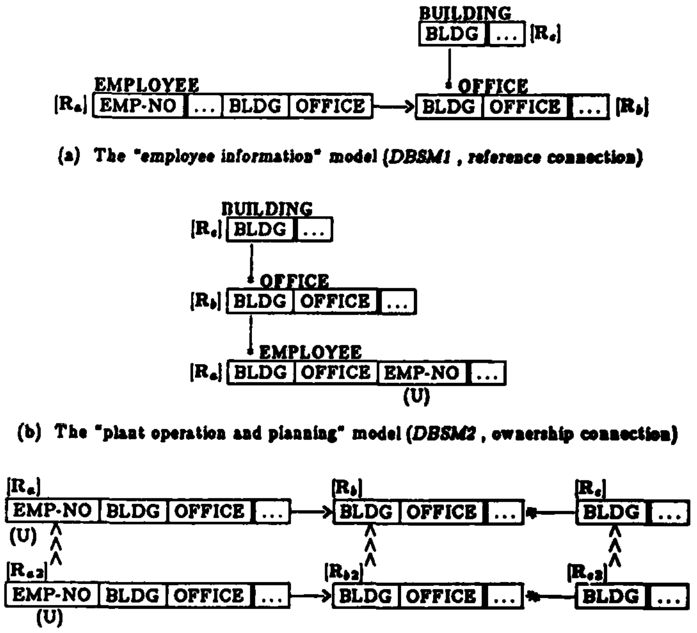
Additional constraints:
dbsm1 :
insert: (1) R_a –: (R_a,(R_{a2})), (2) R_b –: R_b, R_{b2}, (3) R_c –: R_c,R_{c2}.

dbsm2 :
insert: (1) R_a –: (R_a,R_{a2}), (2) R_b –: R_b, R_{b2}, (3) R_c –: R_c,R_{c2}.
delete: (1) R_b –: (R_b),R_{b2}.

This example illustrates how 1:*N* connections are combined when three relation are involved, as discussed in Section 6.4.1.6. In the data models of Figure 53(a) and 52(*b*), *two 1:N connections exist: from* R_c *to* R_b, and *from* R_b to R_a. In Figure 53(a), we have an ownership connection from R_c to R_b, then a reference connection from R_a to R_b (which is 1:*N* from R_b to R_a). The combined connection between R_a and R_c is then a reference connection from R_a to R_c. Hence, each tuple in R_a indirectly references a tuple in R_c that is connected to it via the intermediate tuple in R_b.

In this example, if the OFFICES object class was not represented in *dm2* , then the OFFICE relation would be missing from *dm2* . Hence, we would have to integrate an ownership connection from R_c to R_a in *dm2* with the reference connection from R_a to R_c in *dm1* (Figure 53(a)) that is the combination of the reference connection from R_a to R_b and the ownership connection from R_c to R_b. The resulting integrated database model would be identical to that of Figure 53(c) except for the exclusion of the relation R_{b2}.

In this example, the integration rules applied to the mapping were identical to those of the mechanically obtained rules. In the following examples, the rules are slightly modified to allow for additional semantics of the situation.



(a) The "*employee information*" model (*DBSM1* , *reference connection*)

(b) The "plant operation and planning" model (*DBSM2* , ownership connection)

(c) The integrated database model (*IDBM*)

Figure 53 Integration of two 1:*N* relationships

Example 2:

Consider a hospital database system. Let *dm1* be a "pharmacy inventory" data model, and *dm2* be a "patient information" data model. Consider the relationship PATIENTS:DRUGS between the object classes PATIENTS and DRUGS.

In *dm1* , the relationship describes the current drugs dispensed by the pharmacy, and their relationship to patients that currently use the drugs. Hence, the relationship is a no-dependency, *M:N* relationship, and is represented by an association. This is because once a drug is not being dispensed any more by the pharmacy, no patients will use it. *Figure* 54(a) *shows this model.* A dependent attribute QUANTITY/WEEK is included in the association relation, which gives the quantity of medicine the patient uses of a drug each week.

In *dm2* , the relationship relates each patient to all the drugs he is known to have taken, or is currently taking. The relationship is represented as a nest of references

(Figure 54(b)) since information about a drug should be retained as long as some patients have used or are using the drug. This is also a no-dependency, M:N relationship. Here, the dependent attribute PERIOD gives the number of days the patient has taken the drug (updated weekly), and the dependent attribute CURRENT is a two-valued attribute with value *yes if the patient is currently taking the drug,* and *no* otherwise.

The relationships are not only represented differently, but are also semantically different. Hence, we see that relationships between the same two object classes must be carefully examined when representations differ, even if representations are the same. An exact domain description will help identify such differences. In this example, the domain for the DRUG-ID attribute in *dm1* is "pharmacy drugs", while the domain for the same attribute in *dm2* is "patient drugs".

Hence, we will assume that a semantic analysis is done by the data model designers to define the domains, especially those for ruling part attributes, in a consise manner.

In our current example, the relationship in *dm1* is a subset of that in *dm2*. Hence, R_{ab1} will represent the tuples in R_{ab} of *dm1* in the *idbm*, and R_{ab1} will be a restriction on the attribute CURRENT with value *yes*. This is because the relationship in *dm1* represents the current drugs the patient is taking.

The integrated database model is shown in Figure 54(c). We now give the mapping rules in the notation described above. These rules slightly differ from those given previously for the integration of association and nest of references (see Section 6.4.1.1) to reflect the different domain semantics of the two data models.

Differences between *dm1* and *dm2*:

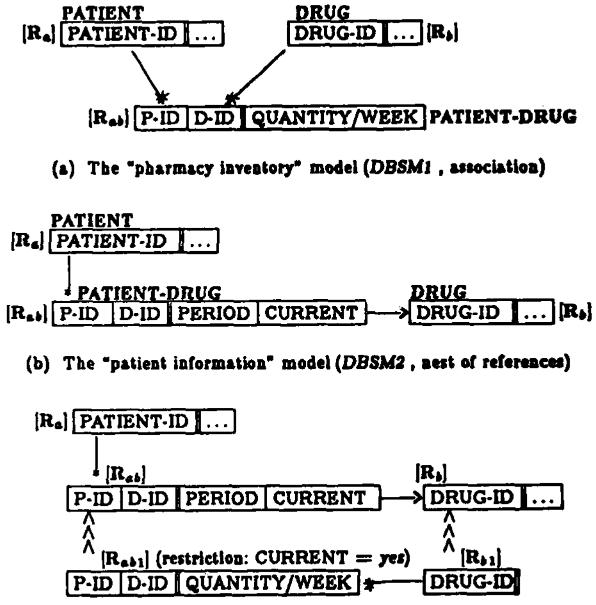
The relationship in *dm1* relates a patient to the drugs he is currently taking, while in *dm2* the relationship relates a patient to the drugs he has taken and is taking.

Additional constraints:
dbsm1:
insert: (1) R_b -: R_b,R_{b1}, (2) R_{ab} -: (R_{ab},R_{ab1}).
delete: (1) R_b -: R_{b1}.
dbsm2:
insert: (1) R_b -: R_b,R_{b1}, (2) R_{ab} -: (R_{ab}).

There are two differences between these rules and the ones previously derived for the integration of association and nest of references in Section 6.4.1.1. The reason for these differences is that the domains of the drugs object classes are different in *dm1* and *dm2*: the DRUGS object class in *dm1* is a subset of that in *dm2*. The differences in the rules are:

(1) When *dbsm2* inserts a tuple in R_{ab}, it is automatically inserted in R_{ab1} if the value of the restricting attribute CURRENT = *yes*.

(2) When *dbsm1* delete a tuple from R_b (some drug if it is no longer in use by the pharmacy), the tuple is only deleted from R_{b1}, and remains in R_b until explicitly deleted by *dbsm2*.



(a) The "pharmacy inventory" model (*DBSM1*, association)

(b) The "patient information" model (*DBSM2*, nest of references)

(c) The integrated database model (*IDBM*)

Figure 54 Integration of two *M:N* relationships with domain differences

The previous rules were derived mechanically, and assumed no additional semantic differences in the domains of the two relations that participate in the relationship. In general, the rules derived mechanically for integrating different representations of a relationship between the same two object classes can serve as a guide to deriving the rules that apply after a concise definition of the domains. If the domains do not differ, the rules are directly applicable.

In the next section, we discuss integration of relationships that have the same cardinality in both data models, but different dependency properties.

6.4.2 Integration of relationships of the same cardinality but different dependency

In this section, we consider the integration of a relationship that is considered to be of the same cardinality by both data models, but has a different dependency in each data model. To be specific, consider a relationship A:B to be of cardinality 1:*N*, and suppose *dm1* considers the relationship to be a no-dependency, while *dm2* considers it a partial depndency of B on A. Then every object in class B in *dm2* must be related to an object from class A, while in *dm1* some objects from class B may exist that are not related to any class A objects. These latter objects cannot exist in *dm2* , since their existence would violate the partial dependency requirement. Hence, the class B objects in *dm2* must be a subset of the class B objects represented in *dm1* .

We now consider some of the different representations of this type in the structural model. In general, when the connections represented in the data models are different, one may have to combine the techniques discussed here with those discussed in the preceding section (section 6.4.1).

6.4.2.1 Three-relation representations in both data models

As a representative of this class, consider the integration of two associations, one a partial and the other a no-dependency. Assume both cardinalities are *M*:*N*. The integration is the same for other cardinalities as long as the cardinality is the same in both data models. Let *dm1* represent the relationship as a no-dependency association, and let *dm2* represent it as a partial dependency (*i*) of A on B.

To represent the partial dependency, a (min i) is specified on the connection from R_a to R_{ab}. First, let us assume that the DLT rule is specified on the (min *i*) constraint. Figure 55 shows this case. The DLT rule specifies that if a tuple is deleted from R_{ab} that causes a violation of the (min *i*) constraint, the tuples causing the violation will also be deleted. We will consider the PD rule later (Figure 56).

Differences:

In *dm2* , every R_a tuple must be connected to at least *i* tuples in R_{ab} (and hence *i* tuples in R_b).

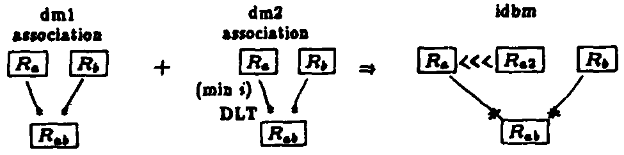


Figure 55 Integration of a no dependency and a partial dependency (*i*) association

Additional constraints:
dbsm1 :
insert: (1) R_{ab}–: (R_{ab},(R_{a2})).
delete: (1) R_{ab}–: R_{ab},(R_{a2}).
dbsm2 :
insert: (1) R_a–: (R_a,R_{a2}).
delete: (1) R_{ab}–: R_{ab},(R_{a2}).

In this case, the subrelation R_{a2} of R_a in the *idbm* will be a restriction subrelation, where the condition for an R_a tuple to belong to R_{a2} is that the tuple be connected to at least *i* tuples in R_{ab}. This restriction is defined on the *idbm* , and is to be automatically enforced when changes are made to the database.

Hence, when *dbsm1* inserts an R_{ab} tuple, if it causes the connected R_a tuple to satisfy the restriction, the R_a tuple is automatically inserted in R_{a2}. For deletion, in the case of the DLT rule being specified on the (min *i*) constraint of *dm2* (Figure 55), if deletion of an R_{ab} tuple results in less than *i* tuples remaining connected to the R_a tuple, that R_a tuple is automatically deleted from R_{a2}, and becomes invisible to *dbsm2* but remains visible to *dbsm1* .

If the PD rule is specified on the (min *i*) constraint of *dm2* , we need an additional relation R_{ab1}, since now if *dbsm1* deletes an R_a tuple that leads to a violation of the (min *i*) constraint of *dbsm2* , it cannot be deleted from *dbsm2* . Hence, the tuple is deleted only from R_{ab1}, and becomes invisible to *dbsm1* . The differences and additional constraints for this case are given below.

Differences:

In *dm2* , every R_a tuple must be connected to at least *i* tuples in R_{ab} (and hence *i* tuples in R_b).

Additional constraints:
dbsm1 :
insert: (1) R_{ab}–: (R_{ab},(R_{a2})).
delete: (1) R_{ab}–: R_{ab1},(R_{ab}).
dbsm2 :
insert: (1) R_a –: (R_a,R_{a2}).

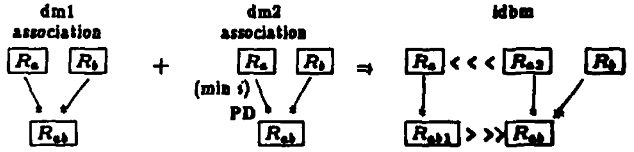


Figure 56 Integration when the PD rule is used to maintain the partial dependency

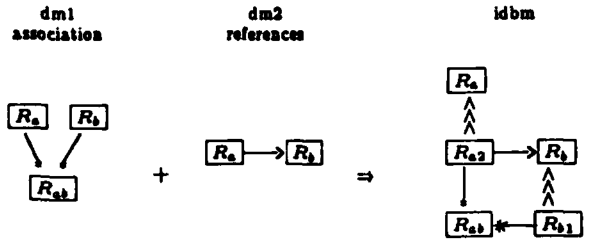


Figure 57 Integration of a 1:N association and a reference

We can similarly develop the integrations when one or both of the connections in the three-relation representation is a *reference*, rather than an *ownership* connection (when a nest of references, or a primary connecting relation is used instead of an association). However, the cases are quite similar to that given above, so we do not consider them here.

Next, we consider the integration of relationships of cardinality 1:N, where one of the relationships is a no-dependency, and the other a partial dependency. The no-dependency relationship is represented by a three- relation representation (association, primary, or nest of references) with its cardinality restricted to 1:N. The partial dependency relationship can also be represented by a three-relation, 1:N, representation, but the integration then is similar to the previous cases. Hence, we only consider the *partial dependency represented by two relations* (*nest or reference connection*).

6.4.2.2 Association and reference

Differences:

(1) In *dm2* , every R_a tuple must reference an R_b tuple, while in *dm1* not all R_a tuples have to be associated with R_b tuples (partial versus no dependency).

(2) In *dm2* , deletion of R_b tuples is restricted by references (reference connection (PD rule) versus ownership connection (DLT rule).

Additional constraints:
dbsm1 :
insert: (1) R_b -: R_b,R_{b1}, (2) R_{ab} -: (R_{a2},R_{ab}).
delete: (1) R_b -: (R_b),R_{b1}.
dbsm2 :
insert: (1) R_a -: (R_a,R_{a2},(R_{ab})), (2) R_b -: R_b,R_{b1}.

The partial dependency requirement that every R_a tuple must reference an R_b tuple in *dm2* leads to the creation of the subrelation R_{a2}. The unrestricted deletion of

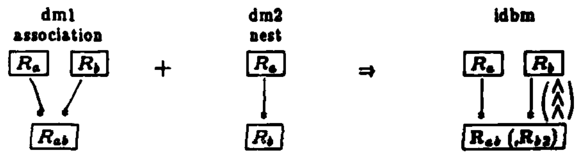


Figure 58 Integration of a 1:N association and a nest

R_b tuples in *dm1* leads to the creation of R_{b1} in the *idbm* .

6.4.2.3 Association and nest

Differences:

(1) In *dm2* , existence of a tuple in R_b requires the existence of the owner tuple in R_a (partial dependency), while in *dm1* R_b tuples can exist independently (no dependency).

(2) In *dm2* , deletion of a tuple from R_a requires the deletion of the owned tuples in R_b (two-relation representation), while *dm1* does not require these deletions (three-relation representation).

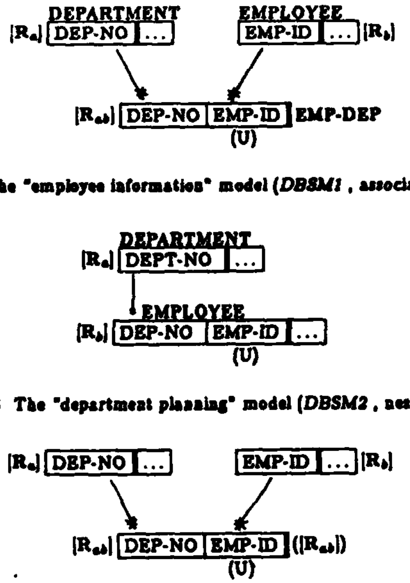
Additional constraints:
dbsm2 :
insert: (1) R_b -: (R_b,R_{b2}).

The R_b tuples in *dbsm2* are only those in R_{b2} in the *idbm* , since these require the existence of the owner tuple. In the *idbm* , R_{ab} will also represent the subset of R_b tuples in R_{b2}.

We give two examples to illustrate this case. In the second example, different ruling part attributes are used in the two data models, which require us to modify one of the data models. This is an example where the two data models could not be integrated directly.

In both examples, *dm1* is a "company information" data model in a dynamic company where departmental structure changes frequently, and employees can exist on the company payroll without being assigned a department. However, the DEPARTMENT: EMPLOYEES relationship is still of cardinality 1:N, but is a no-dependency according to the above description (an employee can exist without being assigned a department).

Data model 2 is a "department planning" database, where departmental changes are planned and executed. Hence, most employee transfers originate from this data model. Here, the DEPARTMENTS object class is the main class, and employees exist only when assigned to a department. Hence, this data model represents the



(a) The "employee information" model (*DBSM1* , association)

(b) The "department planning" model (*DBSM2* , nest)

(c) The integrated database model (*IDBM*)

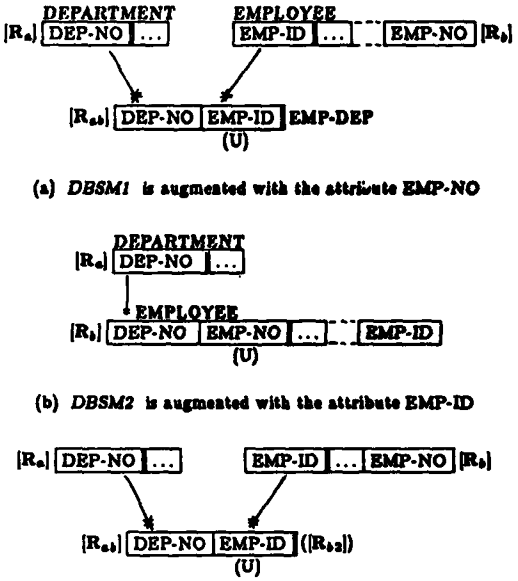
Figure 59 An example of the integration of an association and a nest (same ruling parts)

DEPARTMENT: EMPLOYEES relationship using an ownership connection. When an employee is assigned a department in *dbsm2* , this signals his transfer to a new department.

Example 1:

First, we consider an example where the ruling parts are the same. Here, the attribute EMP-NO identifies the employee in both *dm1* and *dm2* . Since the cardinality of DEPARTMENT:EMPLOYEE is 1:*N*, the EMP-NO attribute must have unique values in tuples of the relations where the attribute is marked (U). Figure 59 shows this example. In this case, the integration is straightforward.

Note how the association relation R_{ab} in the *IDBM* serves as both the connecting relation, as well as to represent the subset of R_b tuples (R_{b2}) of *dbsm2* (Figure 59(c)). This is so because the attribute EMP-NO is unique in the relation.



(a) *DBSM1* is augmented with the attribute EMP-NO

(b) *DBSM2* is augmented with the attribute EMP-ID

(c) The integrated database model (*IDBM*)

Figure 60 Example of integration of an association and a nest (different ruling parts)

Example 2:

In the second example, the ruling part attributes are different in each data mdel. Here, *dm2* uses the two attributes (DEP-NO, EMP-NO) as ruling part, while *dm1* uses a only the single attribute EMP-ID, which is unique over all employees. The attribute EMP-NO has a different value than EMP-ID for the same employee. EMP-NO uniquely identifies an employee within his department only, so employees within different departments can have the same value for EMP-NO.

We must change the two database submodels slightly from the original data models, so that both submodels represent the two attributes EMP-NO and EMP-ID. This is necessary to maintain the correct mapping between the values of the two attributes when new tuples are inserted or identified by either *dbsm1* or *dbsm2* .

Figure 60 shows this example.

We now continue our presentation of the integration of relationships of cardinality

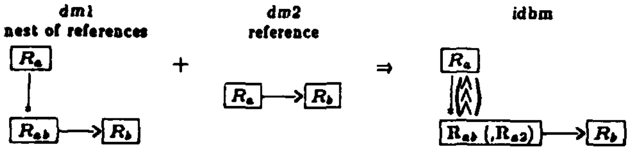


Figure 61 Integration of nest of references and reference (Case 1)

1:*N*, where *one relationship* is a partial dependency that is represented by either an ownership or a reference connection, and the other relationship is a no-dependency. So far we have considered two cases: an association relation representing the no-dependency with a reference connection (Section 6.4.2.2) and with an ownership connection (Section 6.4.2.3). In the next two sections, we consider the integration of with a nest of references representing the *no-dependency*, and in the *following two sections*, we consider the integration with a primary representing the no-dependency.

6.4.2.4 *Nest of references and reference*

Both nest of references and reference are non-symmetric, so we must consider two cases.

Case 1:

Here, we consider the case where the cardinality of the nest of references (R_a, R_b) (see Figure 61) is *N*:1 from R_a to R_b. Since we are currently integrating relationships of the same cardinality, the *reference* connection also has this same cardinality. We will consider the 1:*N* cardinality from R_a to R_b in case 2 below.

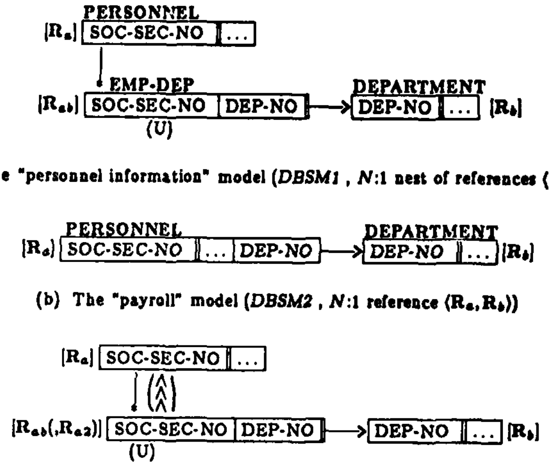
Differences:

A tuple in R_a in *dm2* must be associated with an R_b tuple (partial dependency).

Additional constraints:
dbsm2 :
insert: (1) R_a -: (R_a,R_{a2}).

Example:

Consider a company database. Let *dm2* be the "payroll department" model. A company rule states that every employee on the company payroll must be assigned to a company department, and an employee is a member of only one department. Hence, in *dm2* , a relationship between the object classes PERSONNEL (representing the company employees on the payroll) and DEPARTMENTS (representing the company's departments) is of cardinality *N*:1 and is a partial dependency of PERSONNEL on DEPARTMENTS. The relationship is represented by a reference connection (Figure 62(b)). The connection is a reference (rather than an ownership of EMPLOYEES by DEPARTMENTS) because employee information is as important as department information.



(a) The "personnel information" model (*DBSM1* , *N*:1 nest of references (R_a, R_b))

(b) The "payroll" model (*DBSM2* , *N*:1 reference (R_a,R_b))

(c) The integrated database model (*IDBM*)

Figure 62 Example of integration of an *N*:1 nest of references and a reference

Let *dm1* represent "company personnel" information. Here, all personnel that are related to the company in some way are represented, including those on leaves or just hired (and hence possibly not assigned a department). Hence, the relationship PERSONNEL:DEPARTMENT in *dm1* is represented as an *N*:1 cardinality, no dependency *relationship*, and is represented by an *N*:1 nest of references as shown in Figure 62(a).

Case 2:

In case 1, we considered the integration of a nest of references (R_a, R_b) and a reference (R_a, R_b) where he cardinality was *N*:1 from R_a to R_b. We now consider the integration of the same nest of references (R_a,R_b) with a reference (R_b,R_a) (in the opposite direction to the first case). Here, the cardinality of R_a to R_b is 1:*N* for both representations.

Differences:

(1) Deletion of R_b (R_a) tuples is *restricted in dm1* (*dm2*).

(2) Every R_b tuple in *dm2* must be related to an R_a tuple (partial dependency).

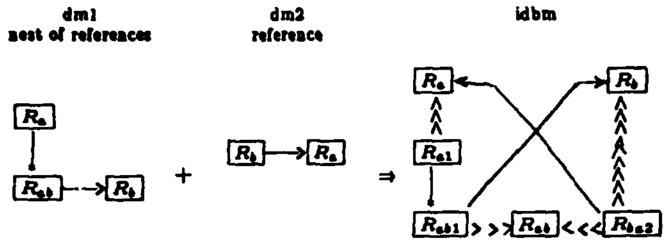


Figure 63 Integration of nest of references and reference (Case 2)

Additional constraints:
dbsm1 :
insert: (1) R_a -: R_a,R_{a1}, (2) R_{ab} -: (R_{ab1}, R_{ab},R_{b2}) .
delete: (1) R_a -: (R_a),R_{a1},(R_{ab}), (2) R_b -: (R_b,R_{ab}), (3) R_{ab} -: R_{ab1}, (R_{ab}).

dbsm2 :
insert: (1) R_a -: R_a,R_{a1}, (2) R_b -: (R_b, R_{b2},R_{ab},R_{ab1}).
delete: (1) R_a -: (R_a,R_{ab}), (2) R_b -: (R_b, R_{ab}),Rb₂.

6.4.2.5 *Nest of references and nest*

Again, both nest of references and nest are non-symmetric, so we must examine two cases.

Case 1:

The cardinality of R_a:R_b is 1:*N*. We integrate a nest of references (R_a, R_b) with a nest (R_a, R_b).

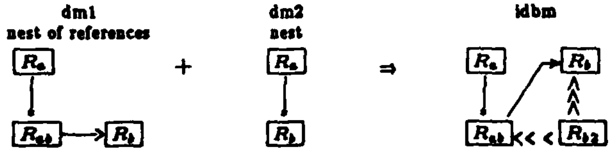


Figure 64 Integration of nest of references and nest (Case 1)

Differences:
(1) R_b tuples may exist independently in *dm1* .
(2) Deletion of R_b *tuples is restricted in dm1* .

Additional constraints:
dbsm1 :
insert: (1) R_{ab} -: (R_{ab},R_{b2}).

dbsm2 :
insert: (1) R_b -: (R_b,R_{ab},R_{b2}).
delete: (1) R_b -: (R_b),R_{b2}.

Case 2:

We integrate a nest of references (R_a, R_b) with a nest (R_b, R_a), in the opposite direction to case 1. The cardinality of R_a:R_b is *N*:1.

Differences:
(1) In *dm1* , R_a tuples can exist independently, while in *dm2* an owner tuple in R_b must exist.
(2) In *dm1* , deletion of R_b tuples is restricted by references, while in *dm1* , deletion of an R_b tuple causes the deletion of the R_a tuples connected to it.

Additional constraints:
dbsm1 :
insert: (1) R_b -: R_b,R_{b2}, (2) R_{ab} -: (R_{ab}, (R_{a2})).

dbsm2 :
insert: (1) R_a -: (R_a,R_{a2},R_{ab}), (2) R_b -: R_b,R_{b2}.
delete: (1) R_b -: (R_b),R_{b2}.

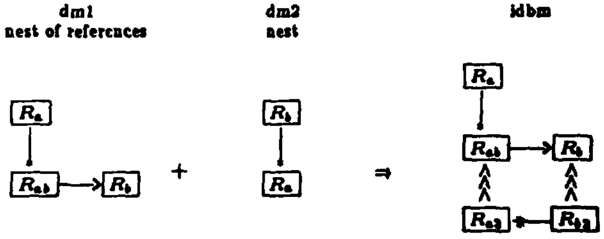


Figure 65 Integration of nest of references and nest (Case 2)

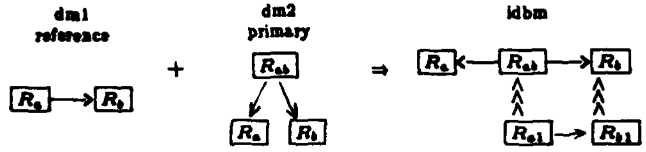


Figure 66 Integration of a 1:*N* primary and a reference

We did not consider examples in the last few cases because they would be similar, more or less, to the ones previously considered. Whenever a case arises where the data models use different representations, the rules presented above can be used, possibly modified by a semantic analysis of the domains of the two data models.

Only two more cases remain which we have not covered wh‿re the cardinality of both relationships is *1:N* (or *N:1*), but the dependencies differ in that one relationship is a partial dependency and the other a no-dependency. These are the cases where one representation is a three- relation with a primary connecting relation, and the other representation is a reference or a nest. We cover these two cases now.

6.4.2.6 Primary and reference

The primary representation is symmetric, so we consider one case only.

Differences:

In dm2 , deletion of R_a *is restricted by references*

Additional constraints:
dbsm1 :
insert: (1) R_b -: R_b, (R_{b1}.
delete: (1) R_b -: (R_{b1}, (R_b)).

dbsm2 :
insert: (1) R_b -: R_b, R_{b1}, (2) R_{ab} -: R_{ab}, R_{a1}.

6.4.2.7 Primary and nest

Differences:

(1) In *dm2* , deletion of R_a is restricted by references

(2) In *dm2* , deletion of an R_a tuple results in deletion of owned R_b tuples, while in *dm2* deletion of R_b is restricted by references

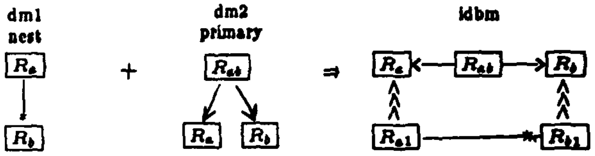


Figure 67 Integration of a 1:*N* primary and a nest

Additional constraints:
dbsm1 :
insert: (1) R_a -: R_a, R_{a1}, (2) R_b -: (R_b, R_{b1}).
delete: (1) R_b -: R_{b1}, (R_b).

dbsm2 :
insert: (1) R_a -: R_a, R_{a1}, (2) R_{ab} -: R_{ab}, R_{b1}.

Finally, we cover the cases where the cardinality is 1:1, and the dependencies are different. For any of the previous cases, if the cardinality of both representations is restricted to 1:1, the integration is the same. Hence, we have only three new cases to cover, where both relationships are represented using two relations only: reference (R_a, R_b) with reference (R_b, R_a), reference (R_a, R_b) with nest (R_a, R_b), and nest (R_a, R_b) with nest (R_b, R_a). In all of these cases, the 1:*N* cardinality in one representation is in the opposite direction to the other representation.

6.4.2.8 Reference and reference

Differences:

(1) In *dm1* (*dm2*), every R_a (R_b) tuple must reference an R_b (R_a) tuple.

(2) Deletion of R_b (R_a) tuples is restricted in *dm1* (*dm2*).

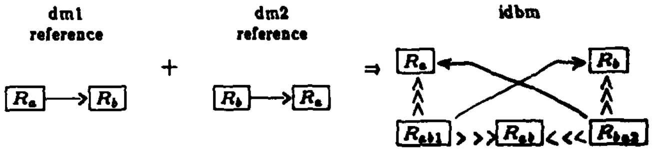


Figure 68 Integration of reference (R_a, R_b) with reference (R_b, R_a)

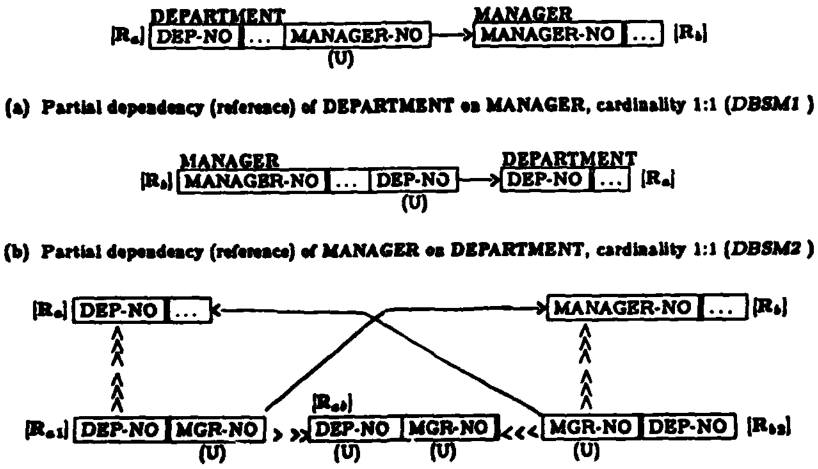
Additional constraints:
dbsm1 :
insert: (1) R_a -: $(R_a,R_{a1},R_{ab},R_{b2})$.
delete: (1) R_a -: $(R_a),R_{a1},(R_{ab})$.

dbsm2 :
insert: (1) R_b -: $(R_b,R_{b2},R_{ab},R_{b2})$.
delete: (1) R_b -: $(R_b),R_{b2},(R_{ab})$.

Here, *dm1* is a partial dependency of R_a on R_b and *dm2* is a partial dependency of R_b on R_a.

Example:

This case is not likely to appear in practice. However, consider a hypothetical situation, where *dm1* represents a 1:1 relationship between MANAGERS and DEPARTMENTS as a partial dependency of DEPARTMENTS on MANAGERS (every department must have a manager). Data model 2 represents the relationship as a partial dependency of MANAGERS on DEPARTMENTS (every manager must manage a department). The two data models are shown in Figures 69(a) and 69(b) respectively.



(a) Partial dependency (reference) of DEPARTMENT on MANAGER, cardinality 1:1 (*DBSM1*)

(b) Partial dependency (reference) of MANAGER on DEPARTMENT, cardinality 1:1 (*DBSM2*)

(c) The integrated database model (*IDBM*)

Figure 69 Example of integration of two 1:1 references

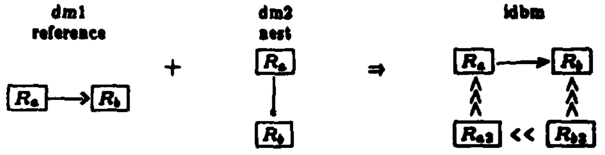


Figure 70 Integration of reference (R_a,R_b) with nest (R_a,R_b)

The integrated database model will then have two relations that represent managers and departments, and two subsets: managers related to a department, and departments that have a manager. The subsets will represent the subclasses of objects that obey the partial dependency constraint of being related to an object of the other class.

6.4.2.9 *Reference and nest*

Differences:

(1) Every R_a tuple in *dm1* must reference an R_b tuple, while in *dm2* every R_b tuple must be owned by an R_a tuple.

(2) Deletion of R_b tuples is restricted by references in *dm1* .

(3) Deletion of an R_a tuple in *dm2* requires deletion of owned R_b tuples.

Additional constraints:
dbsm1 :
insert: (1) R_a -: (R_a,R_{a1},R_{b2}).

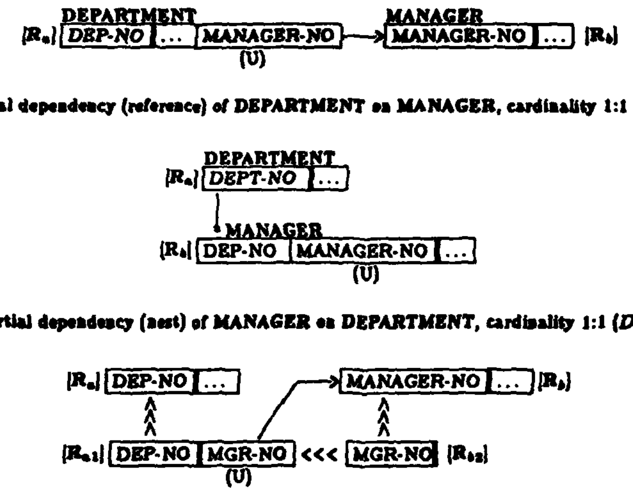
dbsm2 :
insert: (1) R_b -: (R_b,R_{a1},R_{b2}).
delete: (2) R_b -: $(R_b),R_{b2}$.

Example:

Consider the same example discussed of the previous section (the DEPARTMENTS: MANAGERS relationship), and let *dm2* represent the partial dependency by a nest rather than a reference. The example is shown in Figure 71.

6.4.2.8 *Nest and nest*

We finally consider the integration of the case were a 1:1 relationship is represented by different partial dependencies (in *dm1* , the dependency is of R_b on R_a, while in *dm2* the dependency is of R_a on R_b). The cardinality of both representations is restricted to 1:1.



(a) Partial dependency (reference) of DEPARTMENT on MANAGER, cardinality 1:1 (*DBSM1*)

(b) Partial dependency (nest) of MANAGER on DEPARTMENT, cardinality 1:1 (*DBSM2*)

(c) The integrated database model (*IDBM*)

Figure 71 Example of integration of a 1:1 nest and a 1:1 reference

Differences:

(1) In *dm1* (*dm2*), every R_b (R_a) tuple must be owned by an R_a (R_b) tuple.

(2) Deletion of an R_a (R_b) tuple in *dm1* (*dm2*) requires deletion of the owned R_b (R_a) tuple.

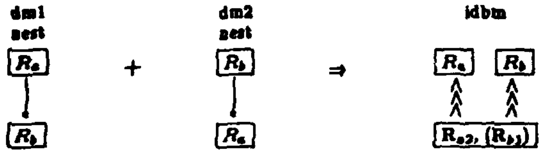
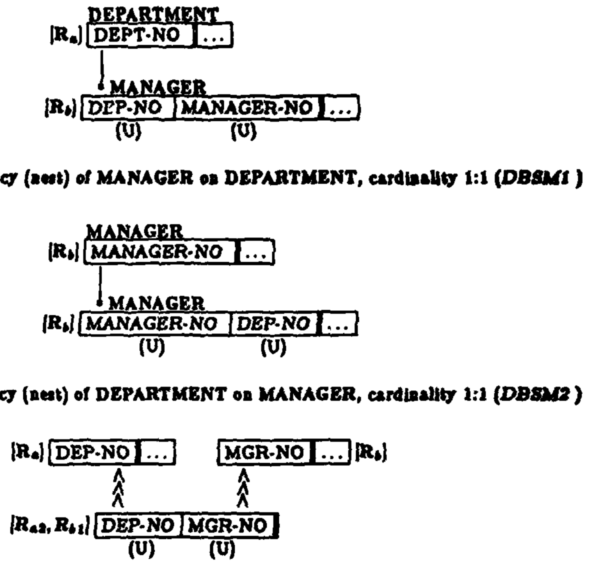


Figure 72 Integration of nest (R_a, R_b) with nest (R_b, R_a)



(a) Partial dependency (nest) of MANAGER on DEPARTMENT, cardinality 1:1 (*DBSM1*)

(b) Partial dependency (nest) of DEPARTMENT on MANAGER, cardinality 1:1 (*DBSM2*)

(c) The integrated database model (*IDBM*)

Figure 73 Example of integration of two 1:1 nests

Additional constraints:
dbsm1 :
insert: (1) R_b –: (R_b,R_{b1}).
dbsm2 :
insert: (1) R_a –: (R_a,R_{a2}).

Example:

We again consider the same example of DEPARTMENTS and MANAGERS. In this case, both connections are ownership connections. Figure 73 shows this example.

We have now considered the possible cases where two relationships have the same cardinality but different dependency properties. In the next section, we consider relationships with different cardinalities.

6.4.3 Integration of relationships of different cardinality

When relationships between the same two object classes have different cardinalities in two data models, the two data models are representing two different relationships. In some cases, when both data models represent the same relationship, but with different cardinality, one of the data models will be in error in the specification of the cardinality of the relationship. In this case, the user group that owns the data model in error will redefine its data model to reflect the correct cardinality of the relationship.

In other cases, the data model with the *more general* cardinality—we define *M*:*N* to be more general than 1:*N*, and 1:*N* to be more general than 1:1—will be representing a relationship that includes the relationship represented in the data model with the more limited cardinality.

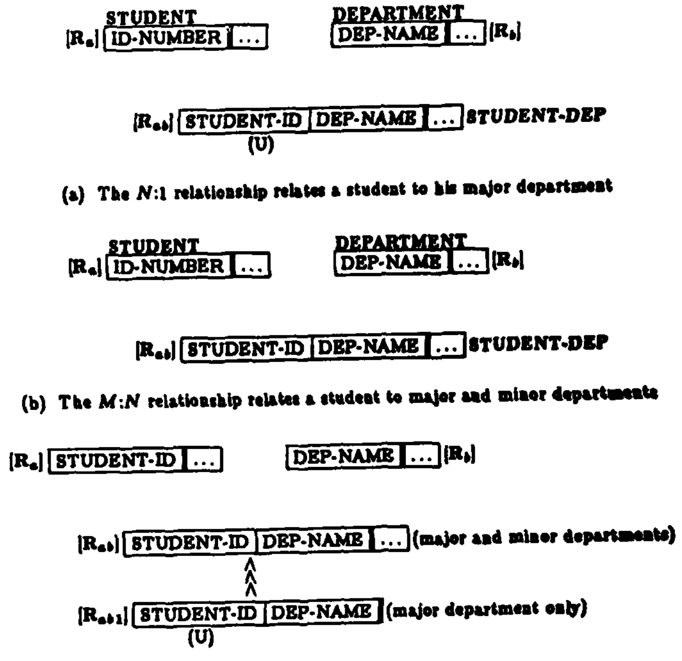
For example, in a university database, the "student directory" data model may *represent the relationship* STUDENTS:DEPARTMENTS to be of cardinality *N*:1, while the "registrar's office" data model may represent the relationship to be of cardinality *M*:*N*. This is explicable *if the student directory lists the major department of* a student only, while the registrar lists the major and minor departments. In this case, the *M*:*N* relationship includes the other 1:*N* relationship as a *subrelationship*.

Definition 24: A relationship REL_1 between two object classes A:B is a subrelationship of a relationship REL_2 between the same two object classes if at all times, whenever two objects $a \in A$ and $b \in B$ are related by REL_1, they are also related by REL_2.

Figure 74 shows this example. It is quite likely in this example that the "registrar's office" data model would inckdude two separate relationships between STUDENTS and DEPARTMENTS: one relationship being *N*:1, which relates a student to his major department, and the other being *M*:*N*, which relates a student to his minor departments (assuming a student can have more than one minor department). Note that if a student can have 0,1, or 2, but not more than two minor departments, the minor department relationship DEPARTMENTS:STUDENTS would be an *M*:N relationship, where *N* is restricted to a maximum of 2.

Note that it is possible that each data model includes only a single relationship between the two object classes, and even give the relationship the same name (for example (students, dept) in the direction from the DEPARTMENT relation). The only difference would be in the cardinality. In Figure 74, the only difference between Figure 74(a) and Figure 74(b) is in the (U) that specifies the 1:*N* cardinality in Figure 74(a).

In general, when cardinalities are different for the same relationship between two objects classes, the data model that represents the relationship *with the less general* cardinality will be representing a subrelationship of the relationship of the more general data model. The *idbm* represents the subrelationship as a subrelation of the relation that represents the general relationship, much in the same way as it would represent a subclass of an object class.


STUDENT
$[R_a]$ ID-NUMBER ...
DEPARTMENT
DEP-NAME ... $[R_b]$
$[R_{ab}]$ STUDENT-ID DEP-NAME ... STUDENT-DEP
(U)
STUDENT
$[R_a]$ ID-NUMBER ...
DEPARTMENT
DEP-NAME ... $[R_b]$
$[R_{ab}]$ STUDENT-ID DEP-NAME ... STUDENT-DEP
$[R_a]$ STUDENT-ID ...
DEP-NAME ... $[R_b]$
$[R_{ab}]$ STUDENT-ID DEP-NAME ... (major and minor departments)
$[R_{ab1}]$ STUDENT-ID DEP-NAME (major department only)
(U)

(a) The *N*:1 relationship relates a student to his major department

(b) The *M*:*N* relationship relates a student to major and minor departments

(c) The integrated database model (*IDBM*)

Figure 74 Integration of a 1:*N* and an *M*:*N* relationship

In Figure 74(c), the relation R_{ab1} represents the restricted relationship. Database submodel 1 (*dbsm1*) only refers to the relation R_{ab1} when relating tuples from STUDENT to DEPARTMENT, and this rule is included in the mapping from *dbsm1* to the *idbm*

6.4.4 Conclusions

In Section 6.4, we considered the integration of two different representations of a relationship. We first considered the case where both the cardinality and dependency were the same (Section 6.4.1), but the representations differed in the connection type. One model used an ownership connection and the other model a reference connection, so the only difference was the way the constraint enforcement is maintained (the ownership connection uses the DLT rule and the reference connection uses the PD rule). From the nature of this difference, we derived the integration rule that the owned relation of the database submodel that uses the ownership connection represents a subrelation of the referencing relation in the submodel that uses the reference connection.

Then we considered the cases where the two representations were of the same cardinality but different dependency. The rule for these cases is that the relation that represents the dependent object class in one database submodel will be a subrelation of the relation that represents an object class that is not dependent in the relationship in the other database submodel.

Finally, we considered the case where the cardinalities were different, and derived the rule that in this case, on relationship is a subrelationship of the other, so that in the *idbm* , the relation that represents the relationship will have a subrelation.

In the next section, we give a procedure for performing the integration.

6.5 THE INTEGRATION PROCEDURE

We now give an algorithm for performing the integration of two data models. This algorithm is not completely general. Rather, it handles only the more common cases. It does not handle connections that are restricted by max and min constraints, or by specifying unique connecting attributes. It does handle the general ownership and reference connections of the structural model without any additional constraints specified on the connections.

When additional constraints on the connections exist, we can resort to the case analysis given in Section 6.4.

We also assume that when some relationship exists in two data models, it is represented in both as having the same cardinality. Our previous assumption that connections are not constrained, along with this assumption, means that the algorithm can only integrate a relationship if it is represented in both data models by the same number of relations (both representations are two-relations or both representations are three-relations, see Section 5.3.1). This is because for unconstrained connections, a two-relation representation is of cardinality 1:*N* and a three-relation representation is of cardinality *M*:*N*.

The special cases can be handled individually, based on the analysis of integration of relationships in Section 6.4.

We do not handle lexicons in this algorithm in order to make the algorithm more comprehensible. Lexicons of ruling parts can be handled in a straightforward manner, since their attribute sets are always interchangeable.

We also assume that all attributes are mandatory.

Input to algorithm:

Two data models *dm1* and *dm2* expressed in the structural model. We assume all domains of the attributes are precisely specified, as discussed in Section 4.3.2.

We use R_i to denote a relation in *dm1* and R_j to denote a relation in *dm2* . We use $K(R_i)$ and $K(R_j)$ to denote the ruling part attributes of R_i and R_j repectively, and $G(R_i)$ and $G(R_j)$ to denote the dependent part attributes of R_i and R_j repectively.

Output of the algorithm:

The *idbm* that supports both data models.

The algorithm:

Procedures used (defined in detail below):

(1) procedure integrate-dep (k, l):

Integrates two relations R_k and R_l that have identical ruling part attributes but differ in their dependent part attributes.

(2) procedure integrate-conn (R_{i1} , R_{i2} , R_{j1} , R_{j2}):

Integrates two connections (ownership or reference), one connection between R_{i1} and R_{i2} , and the other connection between R_{j1} and R_{j2} , where both (R_{i1} , R_{j1}) and (R_{i2} , R_{j2}) are pairs of compatible relations (see Step 1 of the algorithm below).

The algorithm steps:

Step 1 (Find all compatible pairs of relations (R_i , R_j)):

For each pair of relations R_i from dm1 and R_j from dm2 ,

if $DOM(K(R_i)) \bigcap DOM(K(R_j)) \neq \emptyset$, then add ($R_i$, R_j) to a list of compatible pairs of relations.

Step 2 (Integrate the connections):

For every pair of relations R_{i1} and R_{i2} in *dm1* , and R_{j1} and R_{j2} in *dm2* , such that:

(a) (R_{i1} , R_{j1}) and (R_{i2} , R_{j2}) are both in the list of compatible relation pairs, and

(b) A 1:*N* connection (ownership or reference) exists in dm1 from R_{i1} to R_{i2} , and a 1:*N* connection exists *in dm2 from* R_{j1} to R_{j2} .

do the following: call procedure integrate-conn (R_{i1} , R_{i2} , R_{j1} , R_{j2}).

Step 3 (Integrate compatible relations based on ruling and dependent parts):

For every pair of relations (R_i , R_j) in the list of compatible pairs of relations, do either step (a) or step (b).

(a) (Identical domains for ruling part) If $DOM(K(R_i)) = DOM(K(R_j))$, then do either step (a1) or step (a2).

(a1) If $G(R_i) = G(R_j)$, create a single relation R in the *idbm* to support R_i and R_j .

(a2) If $G(R_i) \neq G(R_j)$, create a relation R in the *idbm* that includes the dependent attributes $G(R_i) \bigcap G(R_j)$. Call procedure integrate-dep (j, i). Call procedure integrate-dep (i, j).

(b) (Non-identical domains for ruling part) If $DOM(K(R_i)) \neq DOM(K(R_j))$ (here, $DOM(K(R_i)) \bigcap DOM(K(R_j)) \neq \emptyset$), create a relation R in the *idbm* that includes the dependent attributes $G(R_i) \bigcap G(R_j)$, then do both step (b1) and step (b2).

(b1) If $DOM(K(R_j)) - DOM(K(R_i)) \neq \emptyset$, then create a subrelation $R_{j(idbm)}$ with dependent part attributes $G(R_j) - G(R_i)$; otherwise call procedure integrate-dep (j, i).

(b2) If $DOM(K(R_i)) - DOM(K(R_j)) \neq \emptyset$, then create a subrelation $R_{i(idbm)}$ with dependent part attributes $G(R_i) - G(R_j)$; otherwise call procedure integrate-dep (i, j).

Procedure definitions:

(1) procedure integrate-dep (k, l):

If $G(R_k) - G(R_l) \neq \emptyset$ and the set of attributes $G(R_k) - G(R_l)$ are not all optional in R_k, then create a subrelation $R_{k(idbm)}$ of R in the *idbm* to support R_k, and let $G(R_{k(idbm)}) = G(R_k) - G(R_l)$.

(2) procedure integrate-conn (R_{i1} , R_{i2} , R_{j1} , R_{j2}):

Assume the connecton is 1:*N* in the direction from R_{i1} to R_{i2} , and R_{j1} to R_{j2} . Then we have three cases:

(a) Two *ownership* connections (R_{i1} , R_{i2}) and (R_{j1} , R_{j2}). In this case, two relations R_1 and R_2, and an *ownership* connection (R_1, R_2) are created in the *idbm* .

(b) Two reference connections (R_{i2} , R_{i1}) and (R_{j2} , R_{j1}). In this case, two relations R_1 and R_2, and a reference connection (R_2, R_1) are created in the *idbm* .

(c) One ownership connection (R_{i1} , R_{i2}) and one reference connections (R_{j2} , R_{j1}). In this case, we create relations R_{j2} and R_{j1} , and two subrelations R_{i1} and R_{i2} of R_{j1} and R_{j2} respectively in the *idbm* . We also create a reference connection (R_{j2}, R_{j1}) and an ownership connection (R_{i1}, R_{i2}) in the *idbm* . This case is similar to the case of Section 6.4.1.5.

What the algorithm does is first, for every two compatible relation pairs, connected by a 1:*N* connection, it creates the *idbm* relations and connections based on the differences (if any) between the connections only. Then, it may create further subrelations based on differences between dependent attribute, and between ruling part domains.

6.6 CONCLUSIONS

We see that the process of integration is mainly concerned with the identification of the *relevant subrelations* that *each model is representing*. What we have presented here are the structural aids that can be applied in performing an integration.

We first discussed the different aspects of the process of integration. These aspects were:

(1) Recognition of relations that represent the same object class, discussed in Section 6.3.1. This can be carried out automatically if concise high-level definitions (see Section 4.2) of the domains of ruling parts, and lexicons of ruling parts, of the relations are available.

(2) Examination of (mandatory) dependent part attributes represented in each relation. If relations with compatible ruling part domains have different dependent part attributes, this signals different subclasses of objects represented, which is reflected in the *idbm* by different subrelations of tuples (Section 6.3.2).

(3) Examination of relationships in which the object class participates in both data models. For each relationship between the same two object classes in both data models, different representations may signal different subclasses of objects as discussed in Section 6.4.

(4) Even when all attributes and relationship representations are the same, a quick semantic analysis should be undertaken to discover any differences in the object classes represented in the two data models (Section 6.3.3).

We then gave an *algorithm for performing integration* which is *not* completely general, but covers the common cases (Section 6.5).

An important point to note is that the users never interact with the integrated database model. Hence, the query language is always applied to the database submodel of the user, which the user designed. The complexity of the integrated database model *is completely hidden from the user, and is used by a database management system* to simplify the mapping from a database submodel to the integrated database model.

We also note that although many subrelations may exist for the same base relation in the integrated database model, this is only at the model level. At the implementation level, the base relation and all its subrelations can be in the same file, with a conditional field for each subrelation in each record to indicate whether the record is in the subrelation or not. This is a simple method to reflect the represented subrelations in the integrated database model.

An important consequence for integration, as opposed to the traditional method of defining "views" or "subschemas" over already defined database models, is that integration can provide full update capability for the database submodels. In [DaBe78], it is shown that updatability of relational views is quite limited. In the integration approach, *since we know beforehand what the user desires*, and since the user can specify behavioural properties of structural constraints, we can always provide update

capability. In particular, if two data models are inconsistent with one another, one can always *clarify* the intent of the representation from the users.

The integration procedure covers all the cases where the difference in representation is that of ownership connection versus reference connection, where the connections are not restricted by max and min constraints.

7 CONCLUSIONS AND FUTURE WORK

In this thesis, we presented a new approach to the process of *logical database design*. The traditional approach to logical database design is inherently *centralised*, since the main assumption is that a *database administrator* exists who is responsible for the complete data and processing requirements of some large organisation. The database administrator hence is also assumed to be capable of understanding all users views and providing a totally adequate global database model.

In our approach, each prospective group of users, along with database design experts, are responsible for defining their own processing and data requirements as a data model. These data models are then integrated to form the global database model. This approach is expected to become very important in the future as the trend towards distribution, and away from centralisation, continues.

The five main chapters of this thesis each addressed an important issue in its own right. In Chapter 2, we categorised the important semantic concepts that have been used and proposed for use in database modelling. This categorisation can be useful in the comparison of the numerous modelling approaches that have been proposed. Our categorisation of structural properties of relationships in Section 2.2 is an often neglected aspect of data modelling.

In Chapter 3, we presented our formal tool for database design, the *Structural Model*. The structural model is based on a small number of concepts: domains, attributes, relations, and connections. These limited concepts can correctly represent all the semantic concepts discussed in Chapter 2. It is important to identify a small set of data-structural concepts that are capable of representing a wide variety of semantic concepts, because now the database management system need not be aware of the semantic concepts, but can maintain them based on their structural representation.

The structural model implicitly represents the *structural constraints* of a situation, as well as the methods to maintain these constraints when the underlying database is updated. In Section 3.3, we gave simple algorithms to maintain structural constraints.

In Chapter 4, we turned our attention to the use of the database. We presented two possible languages for use in a database system based on a structural model: the Structural Model Definition Language (SDML) and the Structural Model Query Language (SMQL).

The SMDL is used for data model design, and its statements are chosen so that they can support interactive design of the data model. The design of a data model is an aspect that has often been oversimplified. We believe that a data model should be designed with great care, so that the users will not suffer the consequences of a badly

designed model that they have to deal with for a long period. That is why we provide for statements to modify the data model during the design stage.

The SMQL is based on the functional notation for specifying attributes. However, its tuple selection mechanism, particularly the way *repeating* attributes can be further restricted, is unique as far as we can tell. The tuple selection based on value comparison (relational) operators, as well as on set comparison operators provides additional flexibility, which can be useful for use by higher-level, semantic interfaces, such as natural language or other Artificial Intelligence interfaces.

The SMQL also provides powerful facilities for defining derived structures and update transactions.

In Chapter 5, we showed how to design a data model in the structural modelling approach. We discussed how each of the concepts of Chapter 2 can be represented in the structural model. Then, we presented a systematic, though not completely formal, methodology for data model design. The main concept introduced was that of separating the *basic* data model, which represents the constraints of the situation being modelled, from the *augmented* data model, which includes derived structures to support straightforward user interaction.

Finally, in Chapter 6, we presented the methodology for data model integration. This is the first presentation of data model integration, although the value of integration has often been cited. We categorised the ways in which data models can differ, but still be supported by a common database model. An important consequence of integration, as opposed to the traditional "view" or "subschema" mechanisms, is that complete update privileges can be supported by the database submodels.

We have addressed only one aspect of database design, that of *logical* database design. Another important aspect is the actual *physical* design of the database on computer storage structures.

While logical database design is involved with specifying important constraints, and providing the user with the interface he or she desires, physical database design is involved with performance issues, and can be more complex than logical database design. An important continuation of this work would be to investigate the application of a structural model as an interface between the user and the physical storage structures. This involves a precise definition of possible mappings from the structural model constructs to the file and access methods currently in use for physical database storage.

Another important continuation is to provide a methodology for specifying the expected user access patterns, and performance requirements, on a structural database submodel, and a methodology for integrating these specifications into cummulative expected access patterns and performance requirements on the integrated database model. The methodology can also be extended to include collection of usage statistics on the database submodels, and combining these statistics to identify changing usage patterns which can be used to signal, or even to automatically trigger, a physical database or access structure reorganisation.

Yet another important topic is that of cleanly defining a mapping interface from the integrated database model to the file and access structures, so that if performance requirements trigger a physical database reorganisation, only the mapping interface need be changed.

Several open questions remain with respect to integration. One question is how well the method presented would work when a large number of data models need to be integrated. Another issue would be to attempt to design an algorithm which would take as input two general structural models that do include quantitative constraints. Finally, we need to implement a complete, three-level ANSI/SPARC-like system to test the feasibility aspects of the design methodology.

REFERENCES

[Abr74] Abrial,J.R., "Data Semantics", in Klimbie,J.W. and Koffeman,K.L. (editors), Data Base Management (Proceedings of the IFIP Conference on Data Base Management), North-Holland, 1974, pp.1-60.

[Asea76] Astrahan,M.M., et.al., "System R: Relational Approach to Data Base Management", ACM Transactions on Database Systems, Volume 1, Number 2, June 1976, pp.97–137.

[Bach69] Bachman,C.W., "Data Structure Diagrams", Data Base (Bulletin of the ACM SIGFIDET), Volume 1, Number 2, 1969, pp.4–10.

[BaDa77] Bachman,C.W., and Daya,M., "The Role Concept in Database Models", Proceedings of the Third International Conference on Very Large Data Bases, IEEE, Tokyo, Japan, September 1977, pp.464–476.

[BaPo79] Badal,D.Z., and Popek,G.J., "Cost and Performance Analysis of Semantic Integrity Validation Methods", in Bernstein,P.A. (editor), Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 1979, pp.109–115.

[BeBe79] Beeri,C., and Bernstein,P.A., "Some Computational Problems Related to the the Design of Normal Form Schemata", ACM Transactions on Database Systems, Volume 4, Number 1, March 1979, pp.30–59.

[Bern76] Bernstein,P.A., "Synthesising Third Normal Form Relations from Functional Dependencies", ACM Transactions on Database Systems, Volume 1, Number 4, December 1976, pp.277–298.

[BiNeu78] Biller,H., and Neuhold,E.J., "Semantics of Data Bases: the Semantics of Data Models", Information Systems, Volume 3, Number 1, September 1978, pp.11–30.

[Boea75] Boyce,R.F., et.al., "Specifying Queries as Relational Expressions: the SQUARE Data Sublanguage", Communications of the ACM, Volume 18, Number 11, November 1975, pp.621–628.

[Brad78] Bradley,J., "An Extended Owner-Coupled Set Data Model and Predicate Calculus for Database Management", ACM Transactions on Database Systems, Volume 3, Number 4, December 1978, pp.385–416.

[BrPaPe76] Bracchi,G., Paolini,P., and Pelagatti,G., "Binary Logical Associations in Data Modelling", in Nijssen,G.M. (editor), Modelling in Data Base Management Systems, North Holland, 1976.

[BuFr79] Buneman,P., and Frankel,R.E., "FQL: A Functional Query Language", in Bernstein,P.A. (editor), Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 1979, pp.52–57.

[CaKa78] Carlson,C.R. and Kaplan,R.S., "A Generalised Access Path Model and its Application to a Relational Data Base System", in Rothnie,J.B. (editor), Proceedings of the ACM SIGMOD International Conference on Management of Data, Washington, D.C., June 1978, pp.143–154.

[Cha76] Chang,C.L., "DEDUCE—A Deductive Query Language for Relational Data Bases", in Chen,C.H. (editor), Pattern Recognition and Artificial Intelligence, Academic Press, 1976.

[ChaKe78] Chang,S.-K. and Ke,J.S., "Database Skeleton and its Application to Fuzzy Query Translation", IEEE Transactions on Software Engineering, Volume SE-4, Number 1, January 1978, pp.31–44.

[Chea76] Chamberlin,D.D., et.al., "SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control", IBM Journal of Research and Development, Volume 20, Number 6, November 1976, pp.560–575.

[Chen76] Chen,P.P.-S., "The Entity-Relationship Model—Toward a Unified View of Data", *ACM Transactions on Database Systems*, Volume 1, Number 1, March 1976, pp.9–36.

[CODASYL71] Committee on Data System Languages, CODASYL Data Base Task Group Report, ACM, New York, 1971.

[CODASYL74] CODASYL Data Description Language, Journal of Development (June 1973), National Bureau of Standards Handbook 113, Government Printing Office, Washington, D.C.,January 1974.

[CODASYL78] Committee on Data System Languages, CODASYL Data Base Task Group Revised Report, ACM, New York, 1978.

[CODASYL74] CODASYL Data Description Language, Journal of Development (June 1973), National Bureau of Standards Handbook 113, Government Printing Office, Washington, D.C., January 1974.

[Codd70] Codd,E.F., "A Relational Model for Large Shared Data Banks", Communications of the ACM, Volume 13, Number 6, June 1970, pp.377–387.

[Codd71] Codd,E.F., "A Data Base Sublanguage Founded on the Relational Calculus", Proceedings of the ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, California, November 1971, pp.35–68.

[Codd72a] Codd,E.F., "Relational Completeness of Data Base Sublanguages", in Rustin, R. (editor), Data Base Systems (Courant Computer Science Symposium, Volume 6), Prentice-Hall, 1972, pp.65–98.

[Codd72b] Codd,E.F., "Further Normalisation of the Data Base Relational Model", in Rustin,R. (editor), Data Base Systems (Courant Computer Science Symposium, Volume 6), Prentice-Hall, 1972, pp.33–64.

[Codd74] Codd,E.F., "Recent Investigations in Relational Database Systems", Information Processing 74, IFIP, North-Holland, 1974, pp.1017–1021.

[Coea78] Codd,E.F., et.al., "RENDEVOUS Version 1: An Experimental English Language Query Formulation System for Casual Users of Relational Data Bases", IBM Research Report RJ2144, San Jose, California, January 1978.

[DaBe78] Dayal,U., and Bernstein,P.A., "On the Updatability of Relational Views", Proceedings of the Fourth International Conference on Very Large Data Bases, IEEE, Berlin, West Germany, October 1978, pp.368–378.

[DaKa80] Davidson,J., and Kaplan,S.J., "Parsing in the Absence of a Complete Lexicon", Proceedings of the 18th Annual Meeting of the Association for Computational Linguistics, Philadelphia, Pennsylvania, June 1980, (to appear).

[DeBJo77] DeBlasis,J.P. and Johnson,T.H., "Data Base Administration—Classical Pattern, Some Experiences and Trends", *AFIPS*, Volume 46, Proceedings of the National Computer Conference, June 1977, pp.1–7.

[ElWi79] El-Masri,R., and Wiederhold,G., "Data Model Integration Using the Stuctural Model", in Bernstein,P.A. (editor), Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 1979, pp.191–202.

[ElWi80] El-Masri,R., and Wiederhold,G., "Properties of Relationships and their Representation", Proceedings of the National Computer Conference, May 1980, AFIPS, Volume 49, pp.319–326.

[Fag77] Fagin,R., "Multivalued Dependencies and a New Normal Form for Relational Databases", ACM Transactions on Database Systems, Volume 2, Number 3, September 1977, pp.262–278.

[Falk76] Falkenberg,E., "Concepts for Modelling Information", in Nijssen,G.M. (editor), Modelling in Data Base Management Systems, North Holland, 1976.

[FrSi76] Fry,J.P. and Sibley,E.H., "Evolution of Data-Base Management Systems", ACM Computing Surveys, Volume 8, Number 1, March 1976, pp.7–42.

[Har77] Harris,L.D., "User Oriented Database Query with ROBOT Natural Language System", Proceedings of the Third International Conference on Very Large Data Bases, IEEE, Tokyo, Japan, September 1977.

[HaMc78] Hammer,M. and McLeod,D., "The Semantic Data Model: A Modelling Mechanism for Data Base Applications", in Lowenthal,E., and Dale,N.B., (editors), Proceedings of the ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 1978, pp.26–36.

[Heea78] Hendrix,G.G., Sacerdoti,E.D., Sagalowicz,D., and Slocum,J., "Developing a Natural Language Interface to a Complex System", ACM Transactions on Database Systems, Volume 3, Number 2, June 1978.

[HeStWo75] Held,G., Stonebraker,M.R., and Wong,E., "INGRES: A Relational Data Base System", Proceedings of the National Computer Conference, May 1975, AFIPS, Volume 44, pp.1–7.

[Hon72] Honeywell Information Systems, Integrated Data Store (IDS) Reference Manual, BR69, Wellseley, Massachusetts, 1972.

[HoWaYa79] Housel,B., Waddle,V., and Yao,S.B., "Functional Dependency Model for Logical Data Base Design", Proceedings of the Fifth International Conference on Very Large Data Bases, Rio de Janeiro, Brasil, October 1979, pp.194–208.

[IBM75] IBM, Information Management System Virtual Store (IMS/VS) Reference Manual, GH20-1260-3, White Plains, New York, 1975.

[Kent79] Kent,W., "Limitations of Record-Based Information Models", ACM Transactions on Database Systems, Volume 4, Number 1, March 1979, pp.107–131.

[Kim79] Kim,W., "Relational Database Systems", ACM Computing Surveys, Volume 11, Number 3, September 1979, pp.185–212.

[Kle67] Kleene,S.C., Mathematical Logic, Wiley, 1967.

[McMe75] McLeod,D.J., and Meldman,M.J., "RISS: A Generalised Minicomputer Relational Data Base Management System", AFIPS, Volume 44, Proceedings of the National Computer Conference, May 1975.

[McL76] McLeod,D.J., "High Level Expression of Semantic Integrity Assertions in a Relational Data Base System", S.M. Thesis, Technical Report MIT/LCS/TR-165, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1976.

[McL78] McLeod,D.J., "A Semanatic Data Base Model and its Associated Structured User Interface", Ph.D. Thesis, Technical Report MIT/LCS/TR-214, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, February 1978.

[MRI72] MRI Systems Corporation, System 2000 General Information Manual, Austin, Texas, 1972.

[NaSc78] Navathe,S.B., and Schkolnick,M., "View Representation in Logical Database Design", in Lowenthal,E., and Dale,N.B., (editors), ACM SIGMOD International Conference on Management of Data, Austin, Texas, 1978, pp.144–156.

[PaDaYu74] Parsons,R.G., Dale,A.G., and Yurkanan,C.V., "Data Manipulation Language Requirements for Data Base Management Systems", The Computer Journal, Volume 17, Number 2, May 1974, pp.99–103.

[Reis77] Reisner,P., "The Use of Psychological Experimentation as an Aid to Development of a Query Language", IEEE Transactions on Software Engineering, Volume SE-3, Number 3, May 1977.

[RoMy75] Roussopoulos,N., and Mylopoulos,J., "Using Semantic Networks for Data Base Management", in Kerr,D.S. (editor), Proceedings of the First International Conference on Very Large Data Bases, ACM, 1975.

[Sag77] Sagalowicz,D., "IDA: An Intelligent Data Access Program", Proceedings of the Third International Conference on Very Large Data Bases, IEEE, Tokyo, Japan, September 1977, pp.293–302.

[SchSw75] Schmid,H.A. and Swenson,J.R., "On the Semantics of the Relational Model", in King,F.W. (editor), ACM SIGMOD International Conference on Management of Data, San Jose, California, 1975, pp.211–223.

[Sen75] Senko,M.E., "Specification of Stored Data Structures and Desired Output in DIAM II with FORAL", in Kerr,D.S. (editor), Proceedings of the First International Conference on Very Large Data Bases, ACM, 1975.

[Ship79] Shipman,D.E., "The Functional Data Model and the Data Language DAPLEX", in Bernstein,P.A. (editor), Supplement to the Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 1979, pp.1–19.

[Sib76] Sibley,E.H., "The Development of Database Technology", ACM Computing Surveys, Volume 8, Number 1, March 1976, pp.1–7.

[SiKe77] Sibley,E.H., and Kerschberg,L., "Data Architecture and Data Model Considerations", Proceedings of the National Computer Conference, June 1977, AFIPS, Volume 46.

[SmSm77] Smith,J.M. and Smith,D.C.P., "Database Abstractions: Aggregation and Generalization", ACM Transactions on Database Systems, Volume 2, Number 2, June 1977, pp.105–133.

[Ste75] Steel,T.B.,Jr. (Chairman), ANSI/X3/SPARC, Study Group on Data Base Management Systems Interim Report, FDT (Bulletin of the ACM SIGMOD), Volume 7, Number 2, 1975.

[Ston74] Stonebraker,M., "High Level Integrity Assurance in Relational Data Base Management Systems", Technical Report ERL-M473, Electronic Research Laboratory, University of California, Berkeley, California, May 1974.

[TaFr76] Taylor,R.W., and Frank,L.R., "CODASYL Data-Base Management Systems", ACM Computing Surveys, Volume 8, Number 1, March 1976, pp.67–104.

[TsLo76] Tsichritsis,D.C., and Lochovsky,F.H., "Hierarchical Data-Base Management: A Survey", ACM Computing Surveys, Volume 8, Number 1, March 1976, pp.105–124.

[Ull79] Ullman,J.D., Principles of Database Systems, Computer Science Press, 1980.

[Wal78] Waltz,D.L., "An English Language Query Answering System", Communications of the ACM, Volume 21, Number 7, July 1978, pp.526–539.

[WaWe75] Wang,C.P., and Wedekind,H.H., "Segment Synthesis in Logical Data Base Design", IBM Journal of Research and Development, Volume 19, Number 1, January 1975.

[WeWo80] Weeldryer,J., and Wood,W.T., "The Entity-Category-Relationship Data Model", Honeywell Corporate Technology Center Report (to appear).

[Wied77] Wiederhold,G., Database Design, McGraw-Hill, 1977.

[Wied78] Wiederhold,G., "Management of Semantic Information for Databases", Proceedings of the 3rd USA-Japan Computer Conference, AFIPS and IPSJ, San Fransisco, California, 1978, pp.192–197.

[WiEl79a] Wiederhold,G., and El-Masri,R., "A Structural Model for Database Systems", Tecnical Report CS-79-722, Computer Science Department, Stanford University, Stanford, California, February 1979.

[WiEl79b] Wiederhold,G., and El-Masri,R., "The Structural Model for Database Design", in Chen,P.P. (editor), Proceedings of the International Conference on the Entity-Relationship Approach to System Analysis and Design, Los Angeles, California, December 1979, pp.247-267.

[WiFrWe75] Wiederhold,G., Fries,J.F., and Weyl,S., "Structured Organization of Clinical Data Bases", Proceedings of the National Computer Conference, AFIPS, Volume 44, 1975, pp.479–486.

[WoYo76] Wong,E., and Youssefi,K., "Decomposition—A Strategy for Query Processing", ACM Transactions on Database Systems, Volume 1, Number 3, September 1976, pp.223–241.

[Zan76] Zaniolo,C., "Analysis and Design of Relational Schemata", Ph.D. Thesis, Technical Report UCLA-ENG-7669, University of California, Los Angeles, California, 1976.

[Zlo75] Zloof,M.M., "Query by Example", Proceedings of the National Computer Conference, AFIPS, Volume 44, 1975, pp.479–486.

DATE
FILMED

2-8